US010062429B1

(12) **United States Patent**
Borole et al.

(10) **Patent No.: US 10,062,429 B1**
(45) **Date of Patent: Aug. 28, 2018**

(54) **SYSTEM, APPARATUS AND METHOD FOR SEGMENTING A MEMORY ARRAY**

(71) Applicant: **Intel Corporation**, Santa Clara, CA (US)

(72) Inventors: **Bhushan Borole**, Rancho Cordova, CA (US); **Iqbal R. Rajwani**, Roseville, CA (US); **Anupama A. Thaploo**, Folsom, CA (US); **Sunil Nekkanti**, Rancho Cordova, CA (US); **Altug Koker**, El Dorado Hills, CA (US); **Abhisek R. Appu**, El Dorado Hills, CA (US)

(73) Assignee: **Intel Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **15/488,681**

(22) Filed: **Apr. 17, 2017**

(51) **Int. Cl.**

G11C 5/05	(2006.01)
G11C 11/4094	(2006.01)
G06F 3/06	(2006.01)
G11C 11/4074	(2006.01)
G11C 11/4093	(2006.01)
G06F 13/40	(2006.01)
G06F 9/38	(2018.01)

(52) **U.S. Cl.**
CPC ***G11C 11/4094*** (2013.01); ***G06F 3/0604*** (2013.01); ***G06F 3/0659*** (2013.01); ***G06F 3/0673*** (2013.01); ***G06F 9/3867*** (2013.01); ***G06F 9/3887*** (2013.01); ***G06F 13/4068*** (2013.01); ***G11C 11/4074*** (2013.01); ***G11C 11/4093*** (2013.01)

(58) **Field of Classification Search**
CPC ... G11C 7/1051; G11C 7/1078; G11C 7/1057; G11C 7/22; G11C 7/106
USPC 365/63, 189.12, 240
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

2008/0288720 A1* 11/2008 Atwal G06F 12/0895 711/108
2013/0155798 A1* 6/2013 Kajigaya G11C 7/065 365/208
2016/0062947 A1 3/2016 Chetlur et al.

OTHER PUBLICATIONS

Cook, S., "CUDA Programming," CUDA Hardware Overview, Chapter 3, http://dx.doi.org/10.1016/8978-0-12-415933-4.00003-X, Elsevier, Inc. 2013, pp. 37-52.

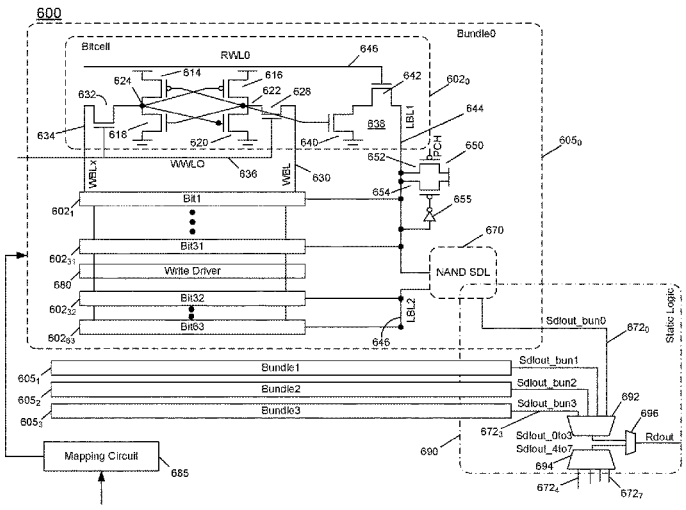
(Continued)

Primary Examiner — Son Dinh
(74) *Attorney, Agent, or Firm* — Trop, Pruner & Hu, P.C.

(57) **ABSTRACT**

In one embodiment, a graphics processor includes a register file having a plurality of storage segments to store information and output a plurality of segment outputs via a plurality of segmented bitlines to a static logic circuit to receive the plurality of segment outputs from the plurality of storage segments and to output read data based on the plurality of segment outputs. The register file may output the read data with a same amount of power without regard to a logic state of the read data. Other embodiments are described and claimed.

20 Claims, 33 Drawing Sheets



(56) **References Cited**

OTHER PUBLICATIONS

Goodfellow, I., et al., "Adaptive Computation and Machine Learning Series," Chapter 5, Nov. 18, 2016, pp. 98-165.
Junkins, S., "The Compute Architecture of Intel Processor Graphics Gen9", Ver. 1, Aug. 14, 2015 (22 pgs.).
Ross, J., et al., "Intel Processor Graphics: Architecture & Programming," PowerPoint presentation, Aug. 2015 (78 pgs).
Wilt, N., The CUDA Handbook: A Comprehensive Guide to GPU Programming, Pearson Education, 2013, pp. 41-57.
U.S. Appl. No. 15/488,687, filed Apr. 17, 2017, entitled "System, Apparatus And Method For Providing A Local Clock Signal For A Memory Array" by Iqbal R. Rajwani, et al..

* cited by examiner

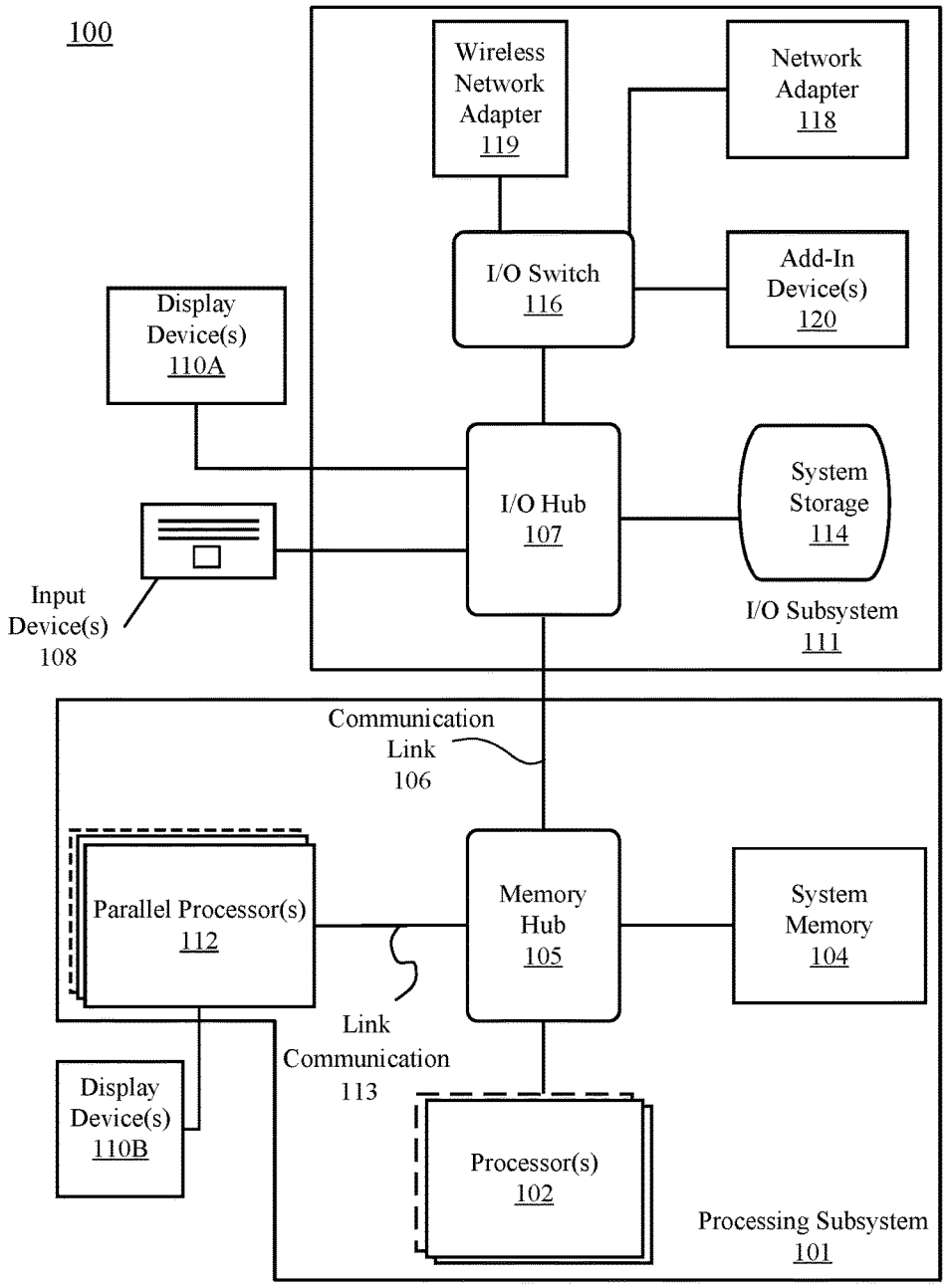

100
Wireless Network Adapter 119
Network Adapter 118
I/O Switch 116
Add-In Device(s) 120
Display Device(s) 110A
I/O Hub 107
System Storage 114
Input Device(s) 108
I/O Subsystem 111
Communication Link 106
Parallel Processor(s) 112
Memory Hub 105
System Memory 104
Link Communication 113
Display Device(s) 110B
Processor(s) 102
Processing Subsystem 101

FIG. 1

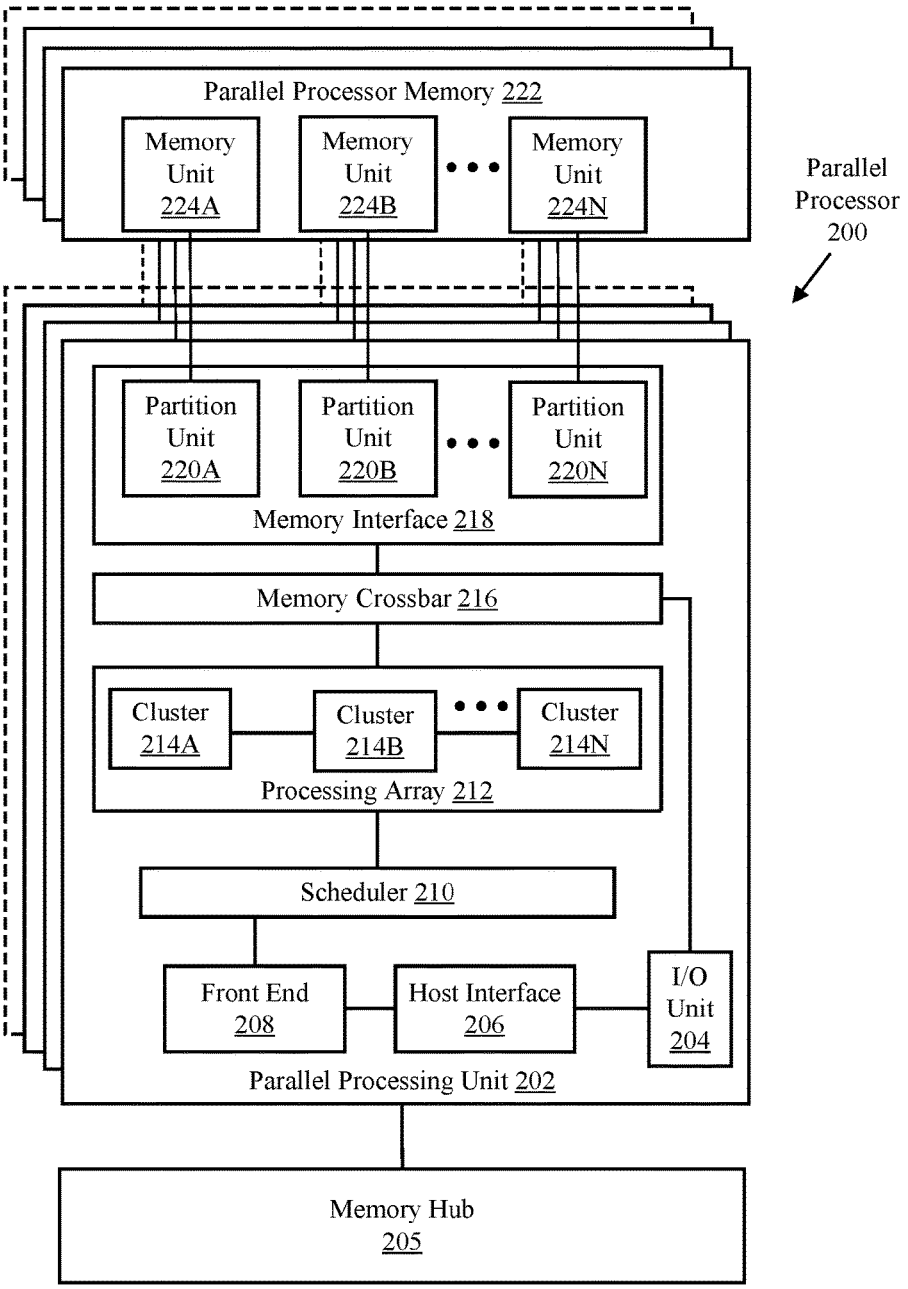

Parallel Processor Memory 222
Memory Unit 224A
Memory Unit 224B
Memory Unit 224N
Parallel Processor 200
Partition Unit 220A
Partition Unit 220B
Partition Unit 220N
Memory Interface 218
Memory Crossbar 216
Cluster 214A
Cluster 214B
Cluster 214N
Processing Array 212
Scheduler 210
Front End 208
Host Interface 206
I/O Unit 204
Parallel Processing Unit 202
Memory Hub 205

FIG. 2A

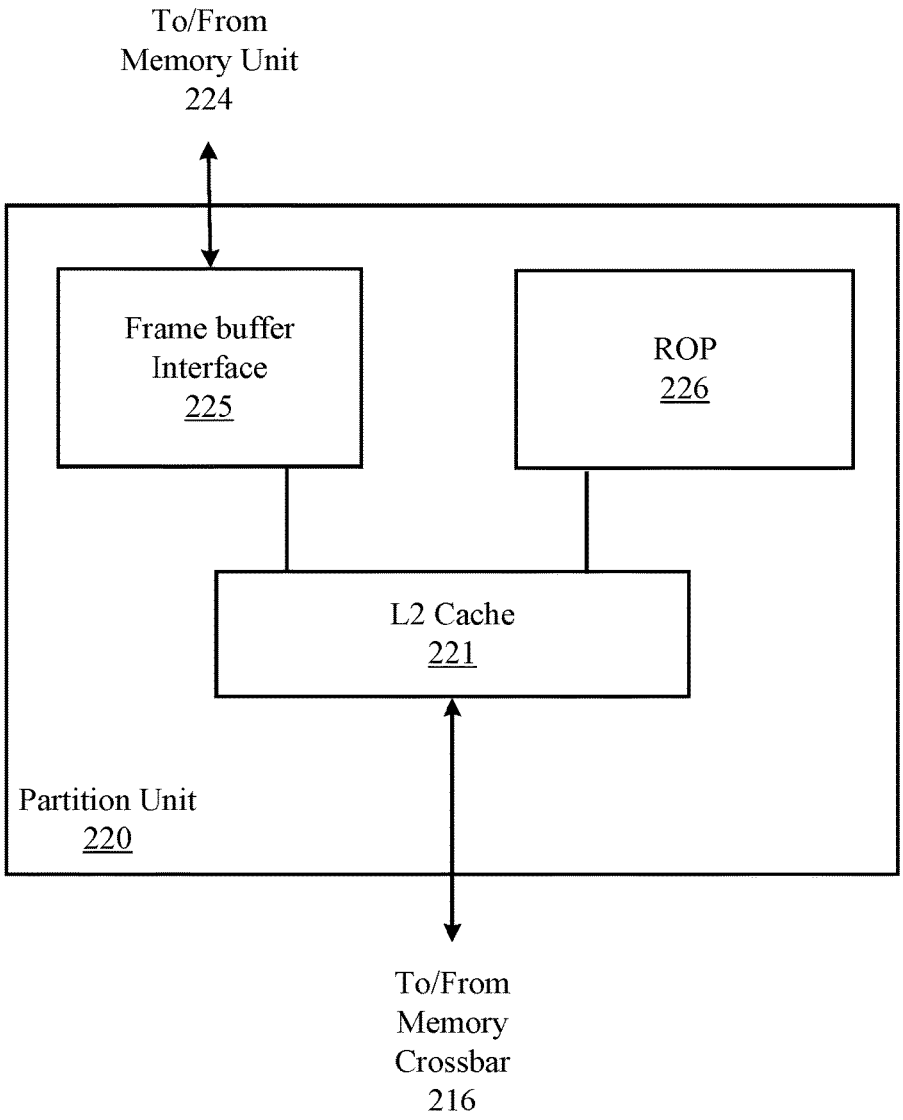

To/From
Memory Unit
224
Frame buffer
Interface
225
ROP
226
L2 Cache
221
Partition Unit
220
To/From
Memory
Crossbar
216

FIG. 2B

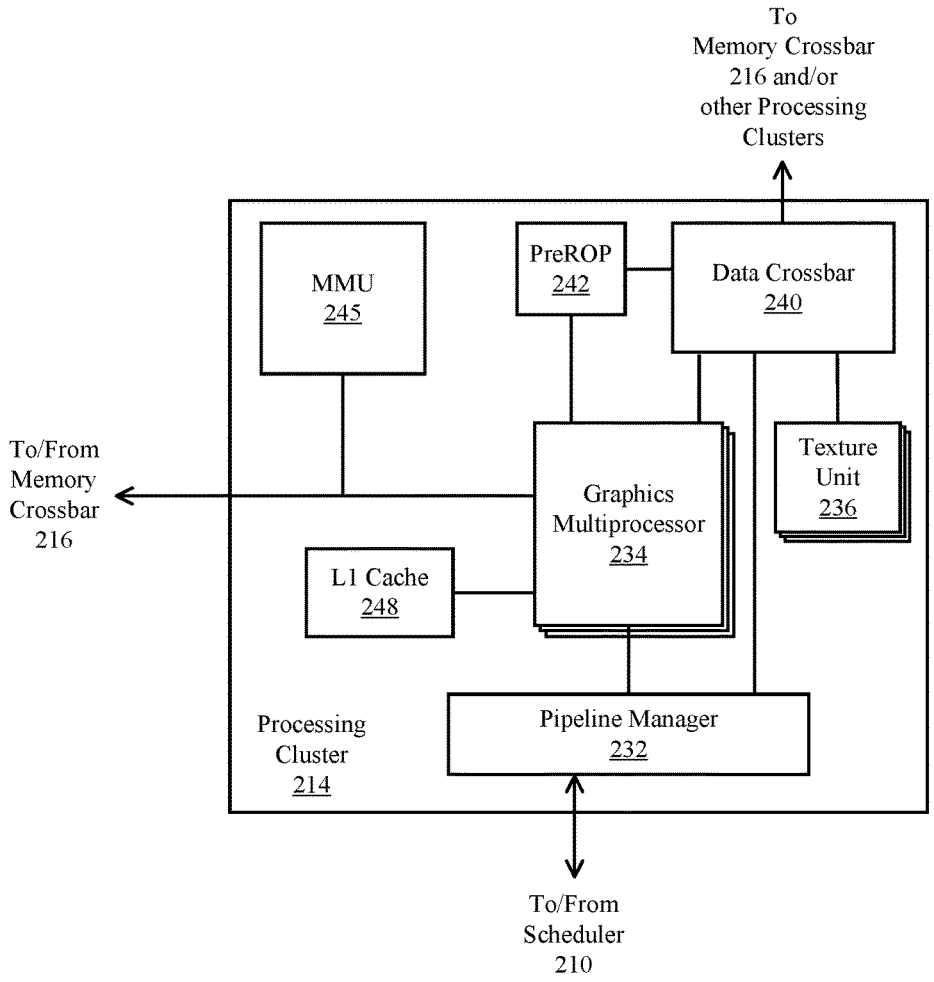

To
Memory Crossbar
216 and/or
other Processing
Clusters
MMU
245
PreROP
242
Data Crossbar
240
To/From
Memory
Crossbar
216
Graphics
Multiprocessor
234
Texture
Unit
236
L1 Cache
248
Processing
Cluster
214
Pipeline Manager
232
To/From
Scheduler
210

FIG. 2C

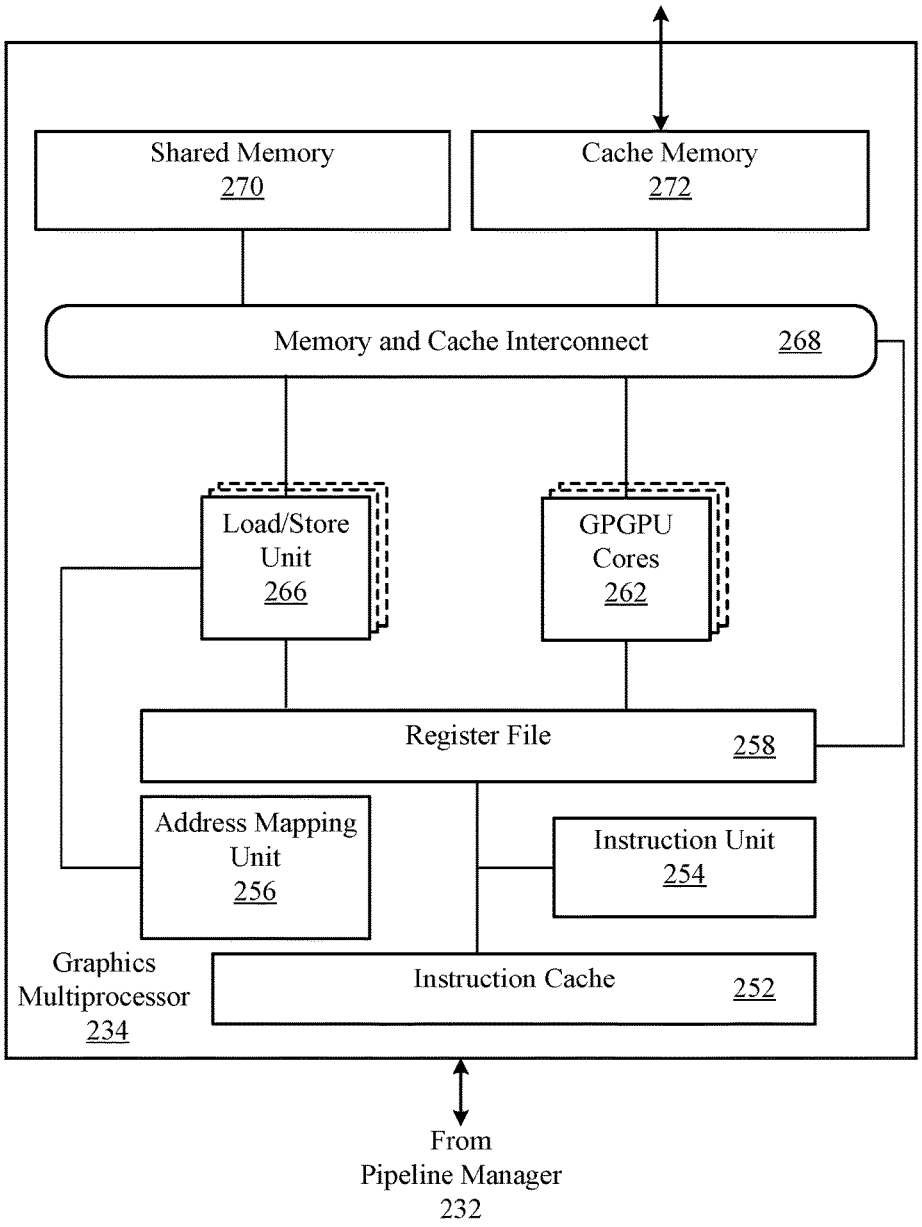

Shared Memory
270
Cache Memory
272
Memory and Cache Interconnect
268
Load/Store
Unit
266
GPGPU
Cores
262
Register File
258
Address Mapping
Unit
256
Instruction Unit
254
Graphics
Multiprocessor
234
Instruction Cache
252
From
Pipeline Manager
232

FIG. 2D

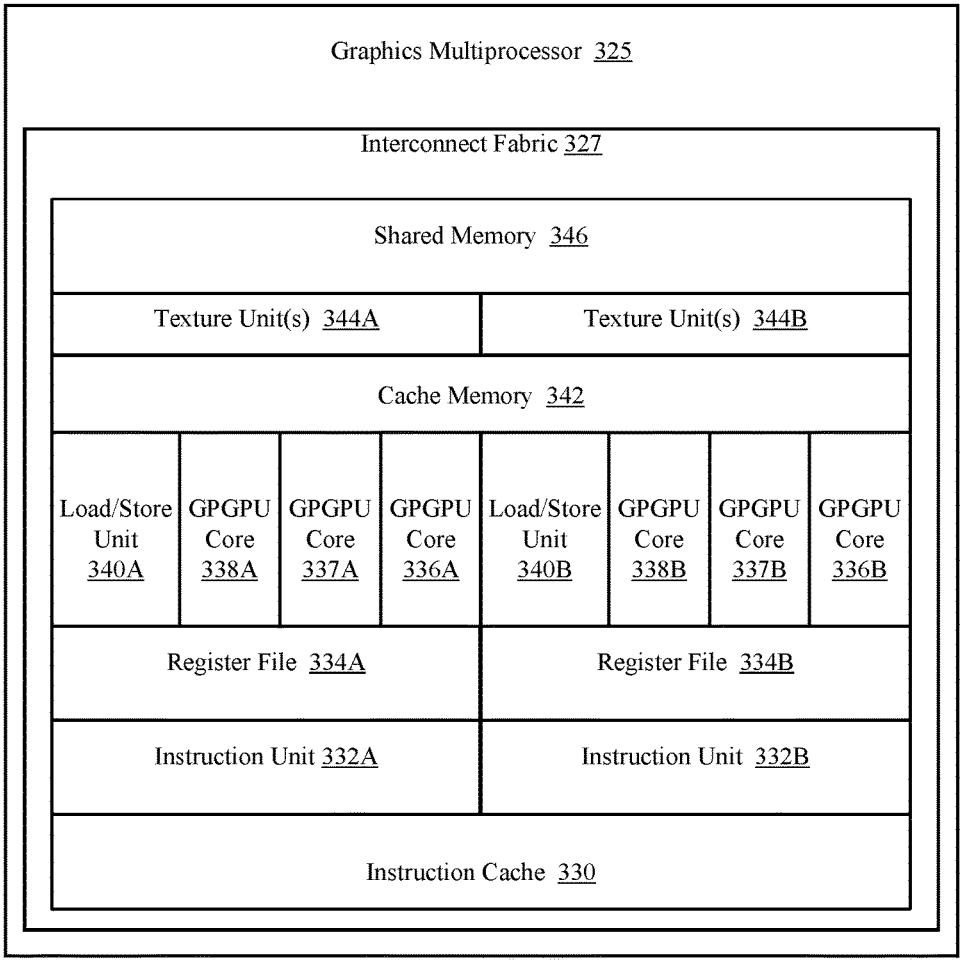

Graphics Multiprocessor 325
Interconnect Fabric 327
Shared Memory 346
Texture Unit(s) 344A
Texture Unit(s) 344B
Cache Memory 342
Load/Store Unit 340A
GPGPU Core 338A
GPGPU Core 337A
GPGPU Core 336A
Load/Store Unit 340B
GPGPU Core 338B
GPGPU Core 337B
GPGPU Core 336B
Register File 334A
Register File 334B
Instruction Unit 332A
Instruction Unit 332B
Instruction Cache 330

FIG. 3A

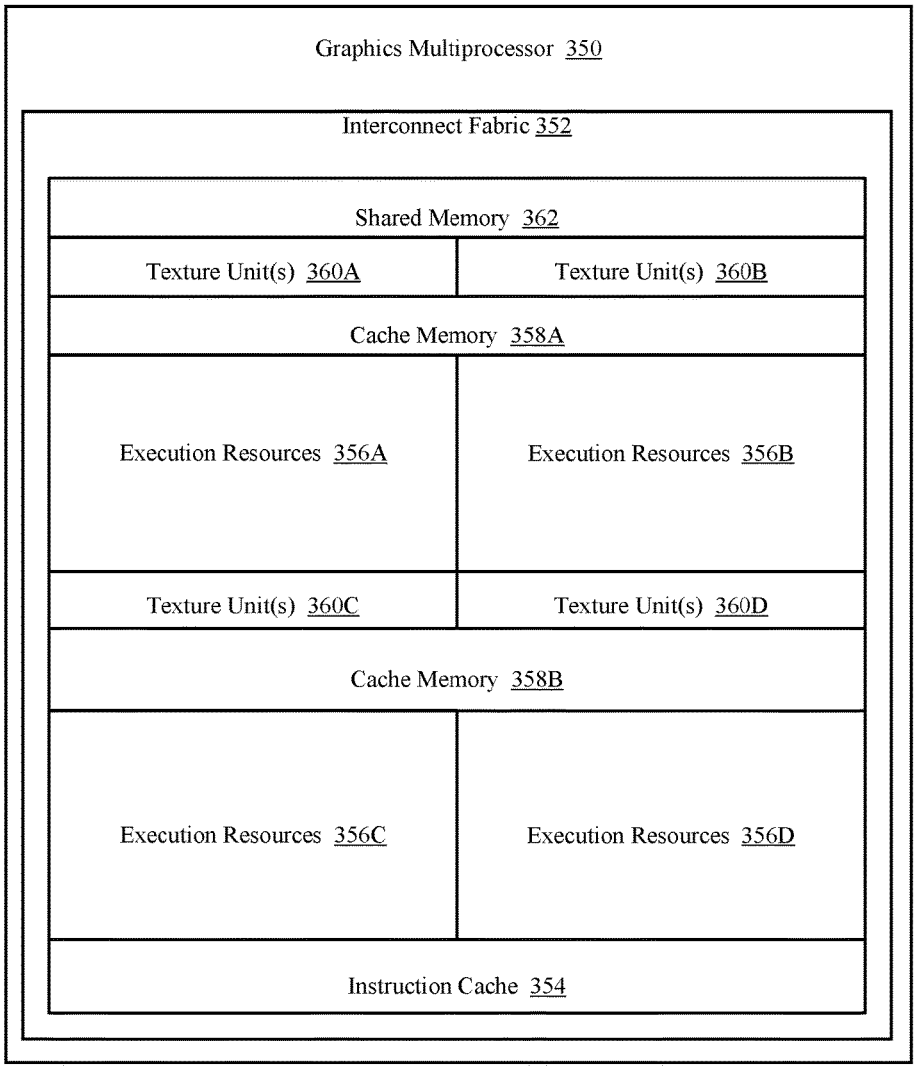

Graphics Multiprocessor 350
Interconnect Fabric 352
Shared Memory 362
Texture Unit(s) 360A
Texture Unit(s) 360B
Cache Memory 358A
Execution Resources 356A
Execution Resources 356B
Texture Unit(s) 360C
Texture Unit(s) 360D
Cache Memory 358B
Execution Resources 356C
Execution Resources 356D
Instruction Cache 354

FIG. 3B

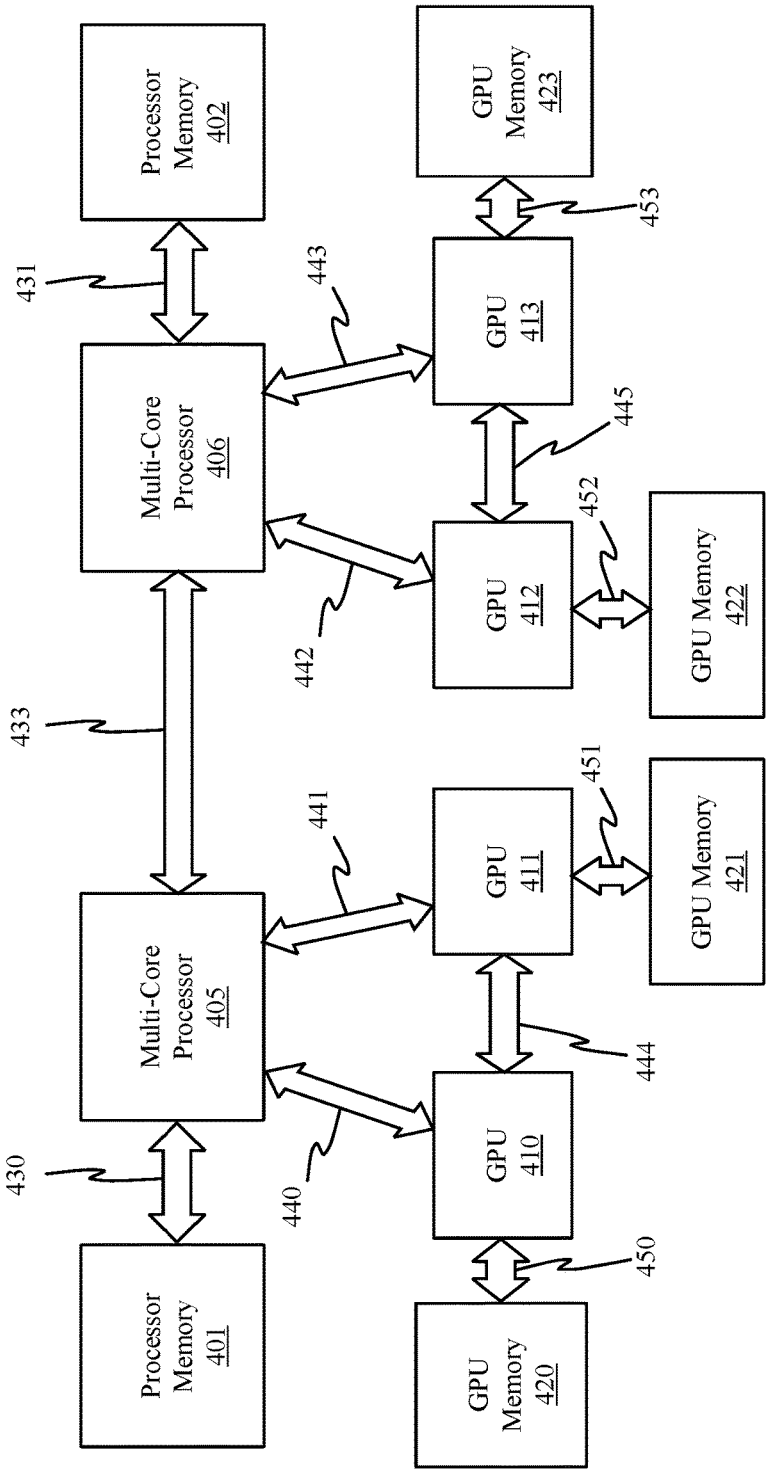

Processor Memory 401
Processor Memory 402
Multi-Core Processor 405
Multi-Core Processor 406
GPU 410
GPU 411
GPU 412
GPU 413
GPU Memory 420
GPU Memory 421
GPU Memory 422
GPU Memory 423
430
431
433
440
441
442
443
444
445
450
451
452
453

FIG. 4A

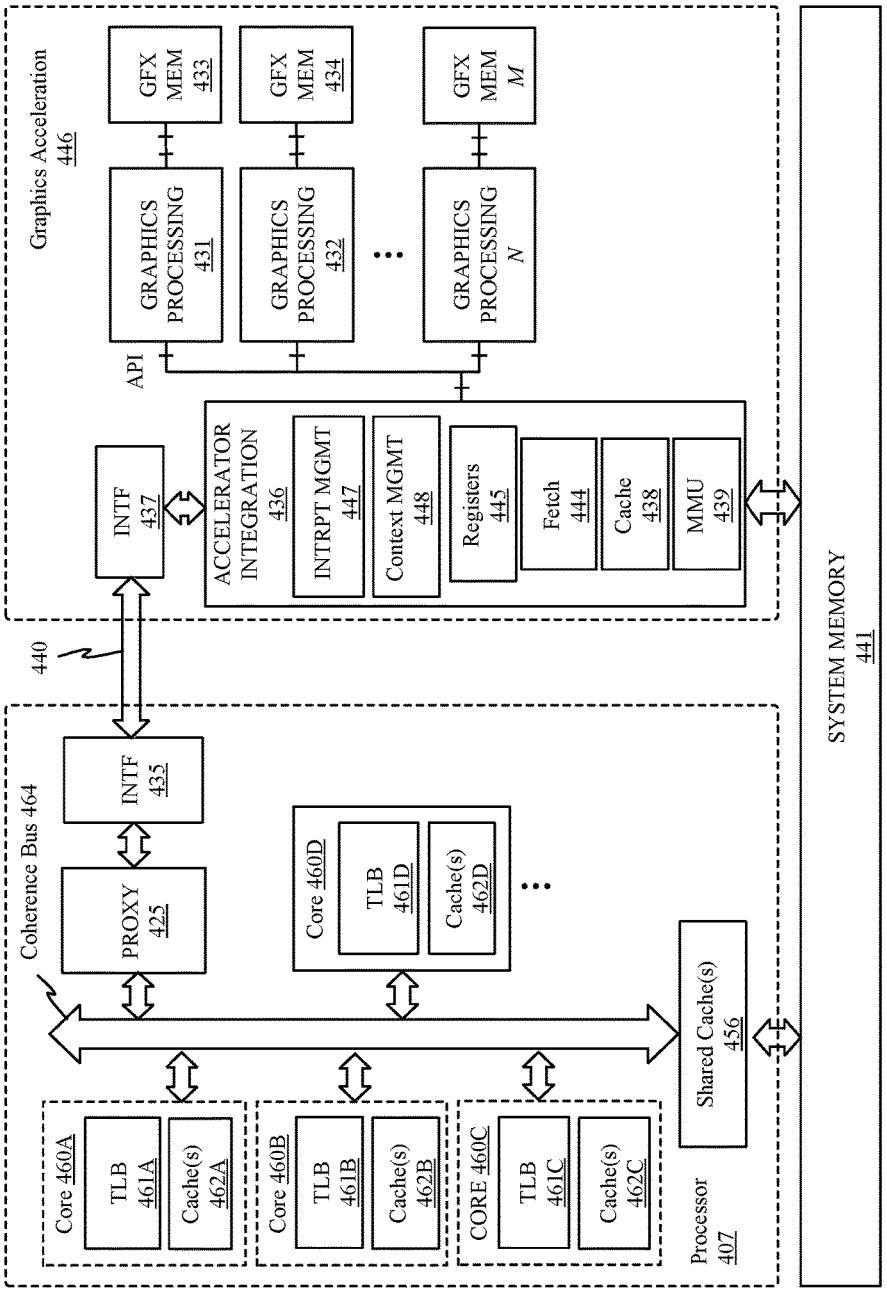

Graphics Acceleration 446
GFX MEM 433
GFX MEM 434
GFX MEM M
GRAPHICS PROCESSING 431
GRAPHICS PROCESSING 432
GRAPHICS PROCESSING N
API
INTF 437
ACCELERATOR INTEGRATION 436
INTRPT MGMT 447
Context MGMT 448
Registers 445
Fetch 444
Cache 438
MMU 439
440
Coherence Bus 464
INTF 435
PROXY 425
Core 460D
TLB 461D
Cache(s) 462D
Core 460A
TLB 461A
Cache(s) 462A
Core 460B
TLB 461B
Cache(s) 462B
CORE 460C
TLB 461C
Cache(s) 462C
Shared Cache(s) 456
Processor 407
SYSTEM MEMORY 441

FIG. 4B

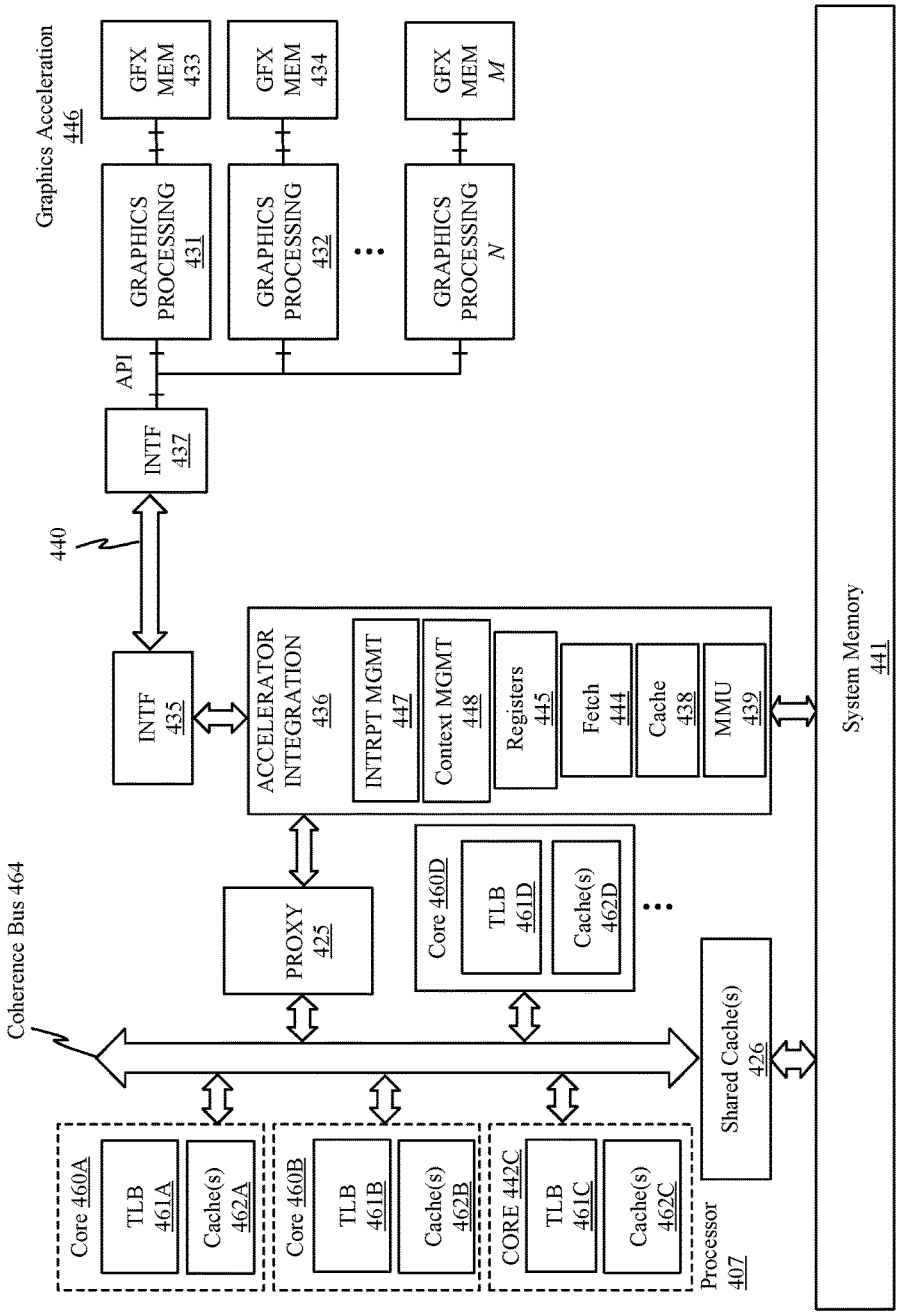

Graphics Acceleration 446
GFX MEM 433
GFX MEM 434
GFX MEM M
GRAPHICS PROCESSING 431
GRAPHICS PROCESSING 432
GRAPHICS PROCESSING N
API
INTF 437
440
INTF 435
ACCELERATOR INTEGRATION 436
INTRPT MGMT 447
Context MGMT 448
Registers 445
Fetch 444
Cache 438
MMU 439
System Memory 441
Coherence Bus 464
PROXY 425
Core 460D
TLB 461D
Cache(s) 462D
Shared Cache(s) 426
Core 460A
TLB 461A
Cache(s) 462A
Core 460B
TLB 461B
Cache(s) 462B
CORE 442C
TLB 461C
Cache(s) 462C
Processor 407

FIG. 4C

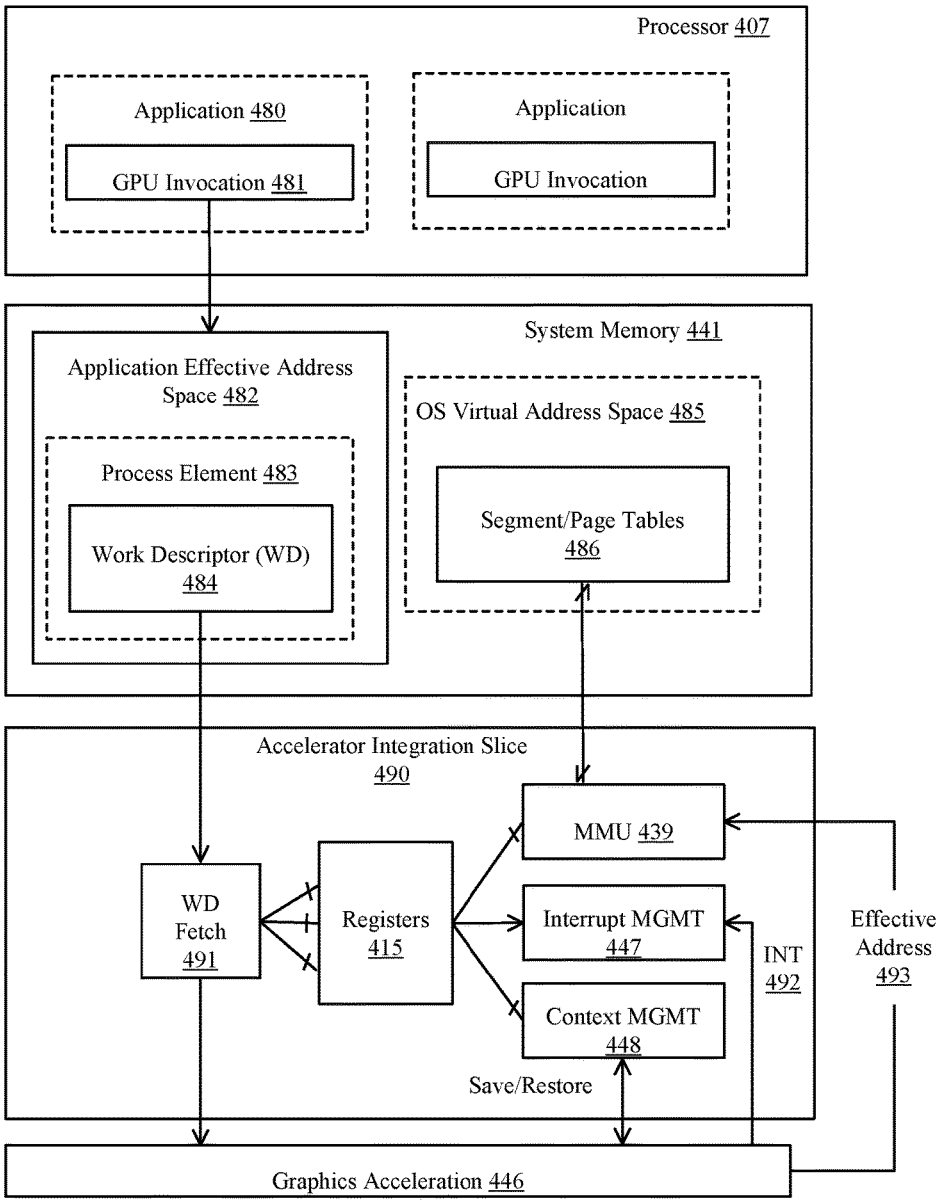

Processor 407
Application 480
GPU Invocation 481
Application
GPU Invocation
System Memory 441
Application Effective Address Space 482
Process Element 483
Work Descriptor (WD) 484
OS Virtual Address Space 485
Segment/Page Tables 486
Accelerator Integration Slice 490
MMU 439
WD Fetch 491
Registers 415
Interrupt MGMT 447
Context MGMT 448
INT 492
Effective Address 493
Save/Restore
Graphics Acceleration 446

FIG. 4D

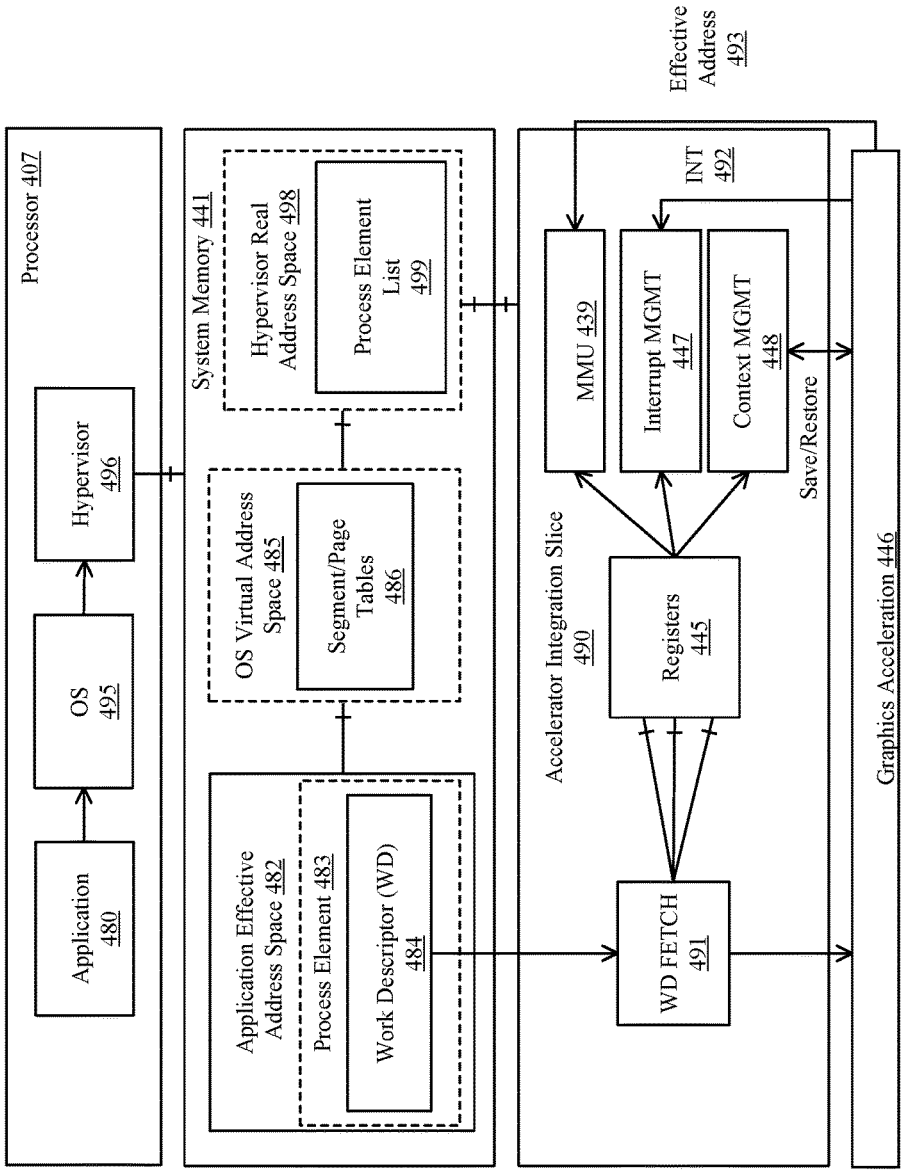

Processor 407
Application 480
OS 495
Hypervisor 496
System Memory 441
Application Effective Address Space 482
Process Element 483
Work Descriptor (WD) 484
OS Virtual Address Space 485
Segment/Page Tables 486
Hypervisor Real Address Space 498
Process Element List 499
Accelerator Integration Slice 490
WD FETCH 491
Registers 445
MMU 439
Interrupt MGMT 447
Context MGMT 448
Save/Restore
INT 492
Effective Address 493
Graphics Acceleration 446

FIG. 4E

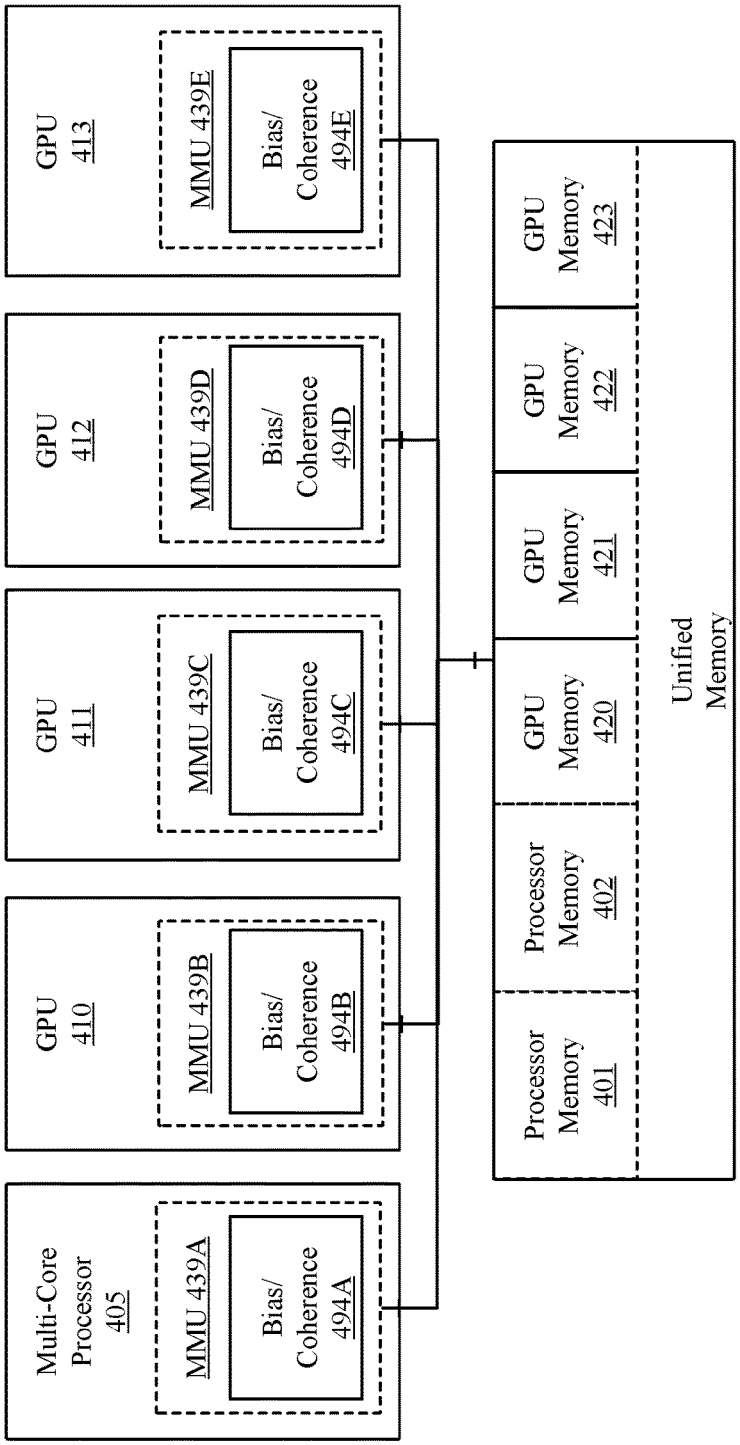

Multi-Core Processor 405
MMU 439A
Bias/ Coherence 494A
GPU 410
MMU 439B
Bias/ Coherence 494B
GPU 411
MMU 439C
Bias/ Coherence 494C
GPU 412
MMU 439D
Bias/ Coherence 494D
GPU 413
MMU 439E
Bias/ Coherence 494E
Processor Memory 401
Processor Memory 402
GPU Memory 420
GPU Memory 421
GPU Memory 422
GPU Memory 423
Unified Memory

FIG. 4F

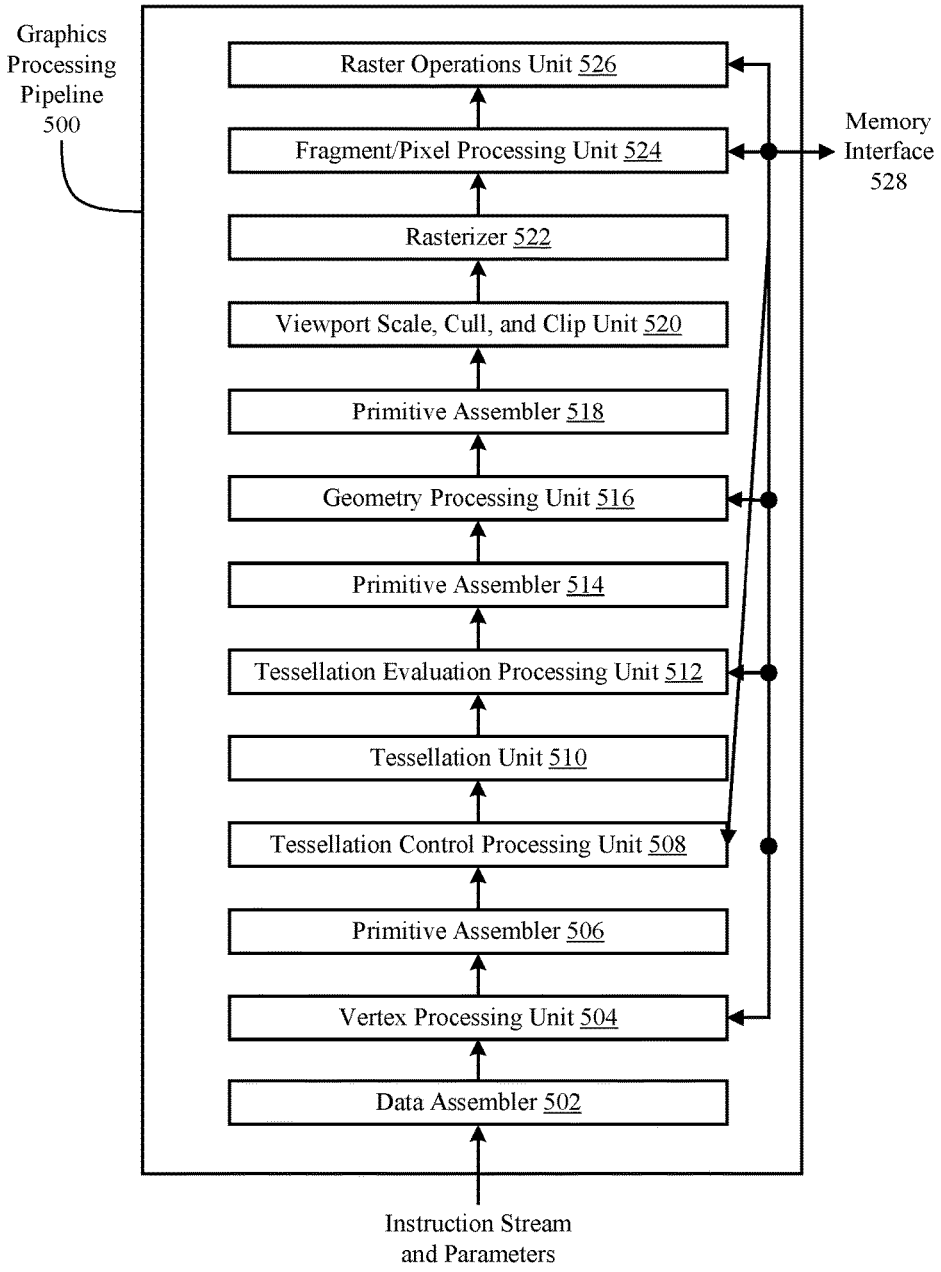

Graphics Processing Pipeline 500
Raster Operations Unit 526
Fragment/Pixel Processing Unit 524
Rasterizer 522
Viewport Scale, Cull, and Clip Unit 520
Primitive Assembler 518
Geometry Processing Unit 516
Primitive Assembler 514
Tessellation Evaluation Processing Unit 512
Tessellation Unit 510
Tessellation Control Processing Unit 508
Primitive Assembler 506
Vertex Processing Unit 504
Data Assembler 502
Memory Interface 528
Instruction Stream and Parameters

FIG. 5

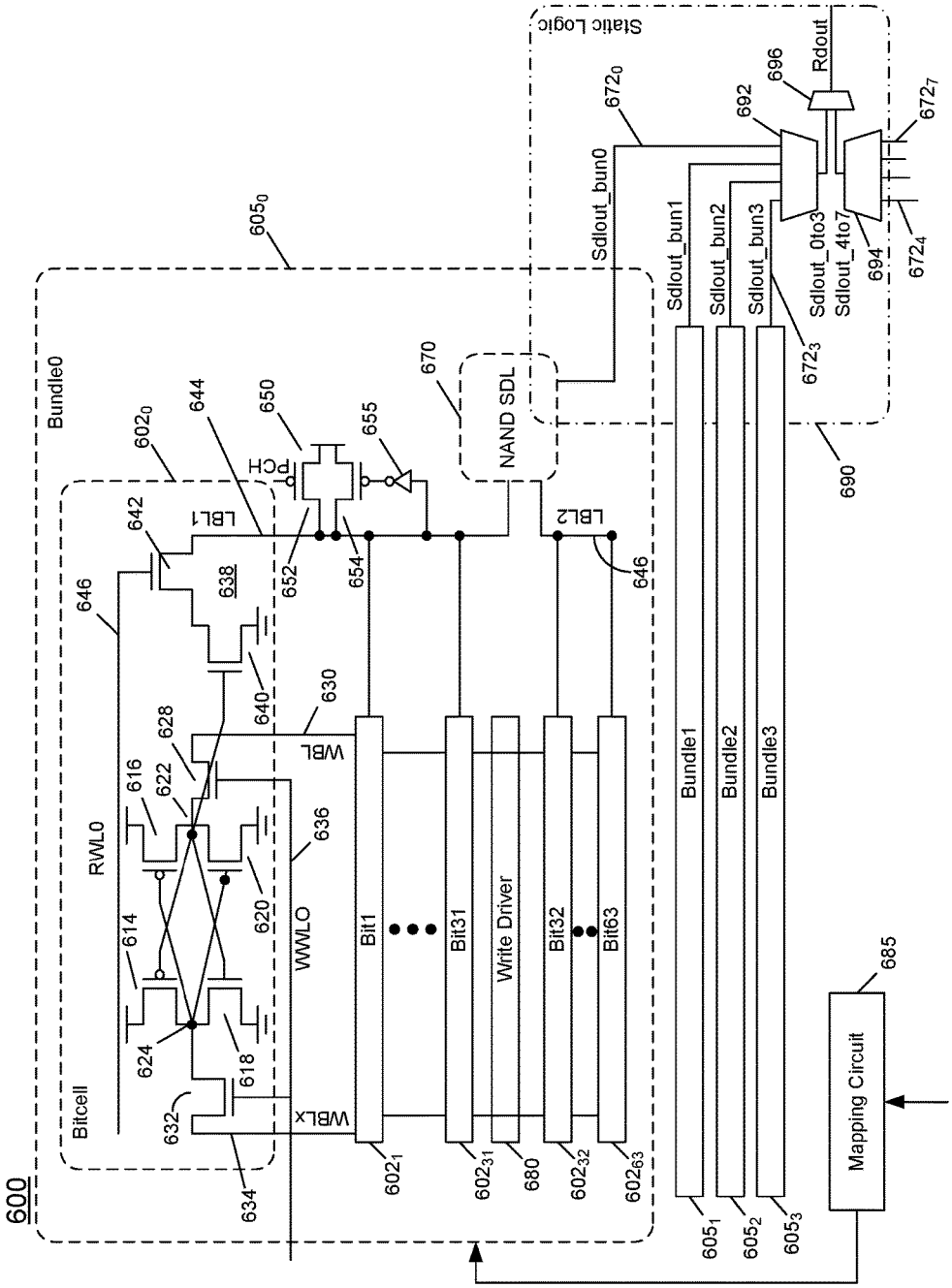

600
Bundle0
Bitcell
RWL0
WWLO
WBLx
WBL
LBL1
LBL2
PCH
NAND SDL
Bit1
Bit31
Bit32
Bit63
Write Driver
Bundle1
Bundle2
Bundle3
Mapping Circuit
Static Logic
Sdlout_bun0
Sdlout_bun1
Sdlout_bun2
Sdlout_bun3
Sdlout_0to3
Sdlout_4to7
Rdout
605_0
602_0
602_1
602_{31}
602_{32}
602_{63}
605_1
605_2
605_3
672_0
672_3
672_4
672_7
614
616
618
620
622
624
628
630
632
634
636
638
640
642
644
646
650
652
654
655
670
680
685
690
692
694
696

FIG. 6

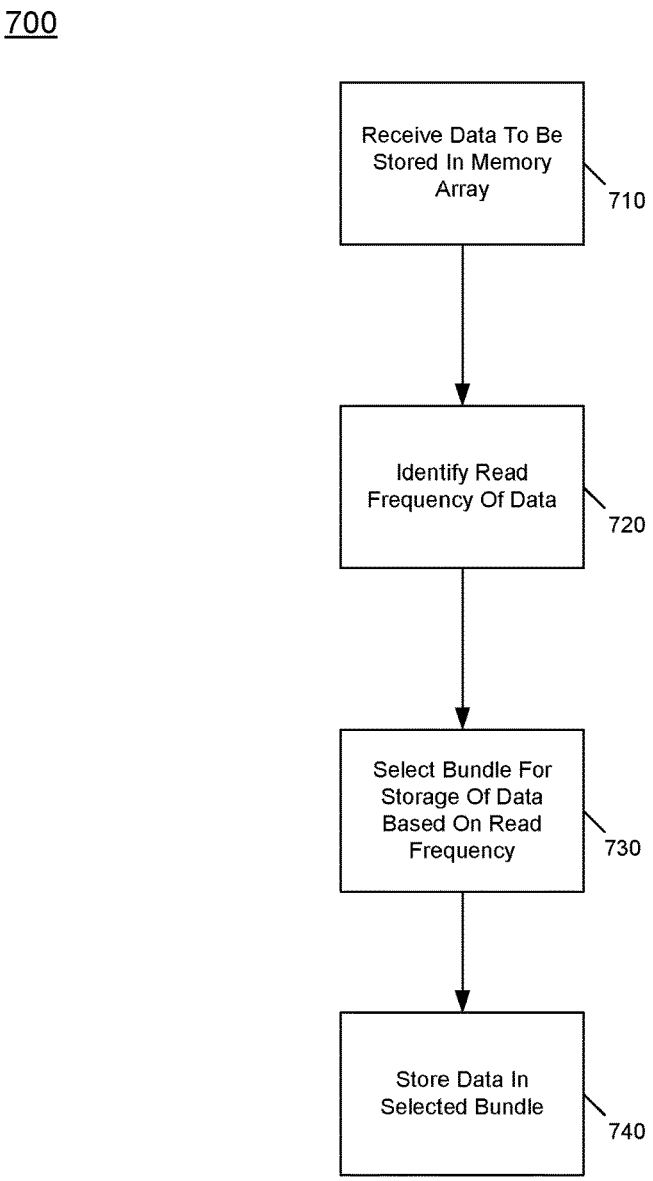

700
Receive Data To Be Stored In Memory Array
710
Identify Read Frequency Of Data
720
Select Bundle For Storage Of Data Based On Read Frequency
730
Store Data In Selected Bundle
740

FIG. 7

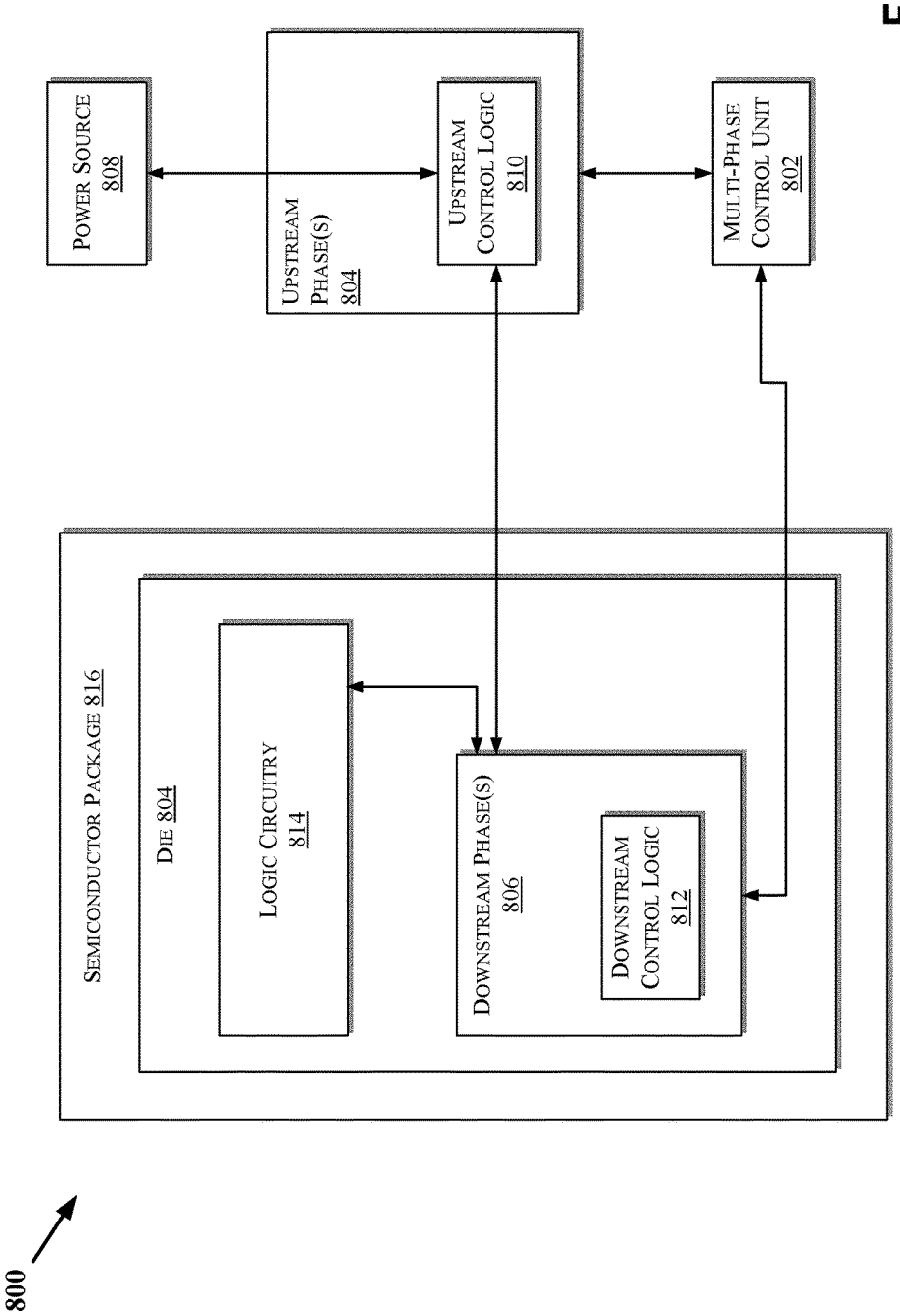

800
Power Source
808
Upstream
Phase(s)
804
Upstream
Control Logic
810
Multi-Phase
Control Unit
802
Semiconductor Package 816
Die 804
Logic Circuitry
814
Downstream Phase(s)
806
Downstream
Control Logic
812

FIG. 8

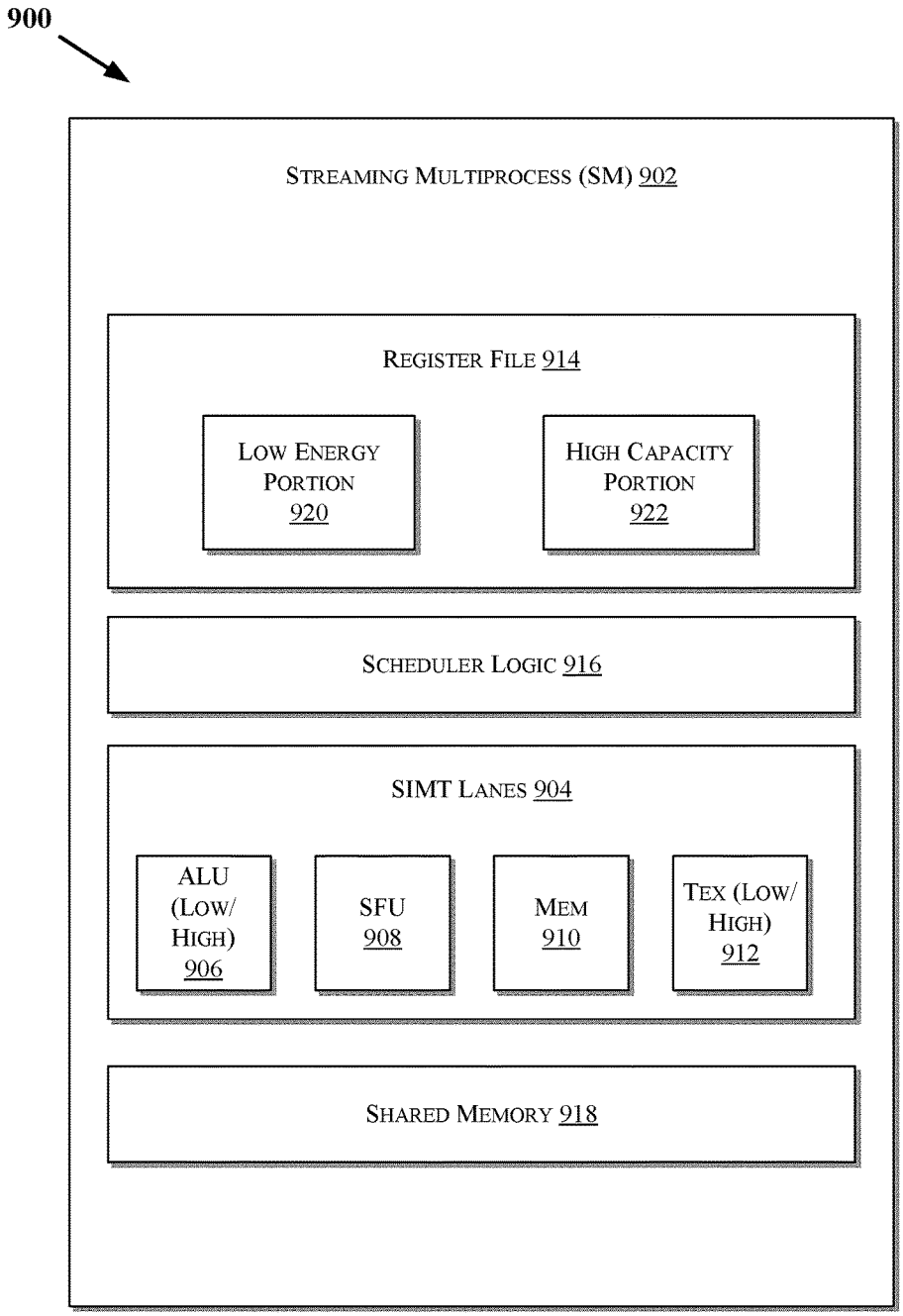

900
STREAMING MULTIPROCESS (SM) 902
REGISTER FILE 914
LOW ENERGY PORTION 920
HIGH CAPACITY PORTION 922
SCHEDULER LOGIC 916
SIMT LANES 904
ALU (LOW/ HIGH) 906
SFU 908
MEM 910
TEX (LOW/ HIGH) 912
SHARED MEMORY 918

FIG. 9

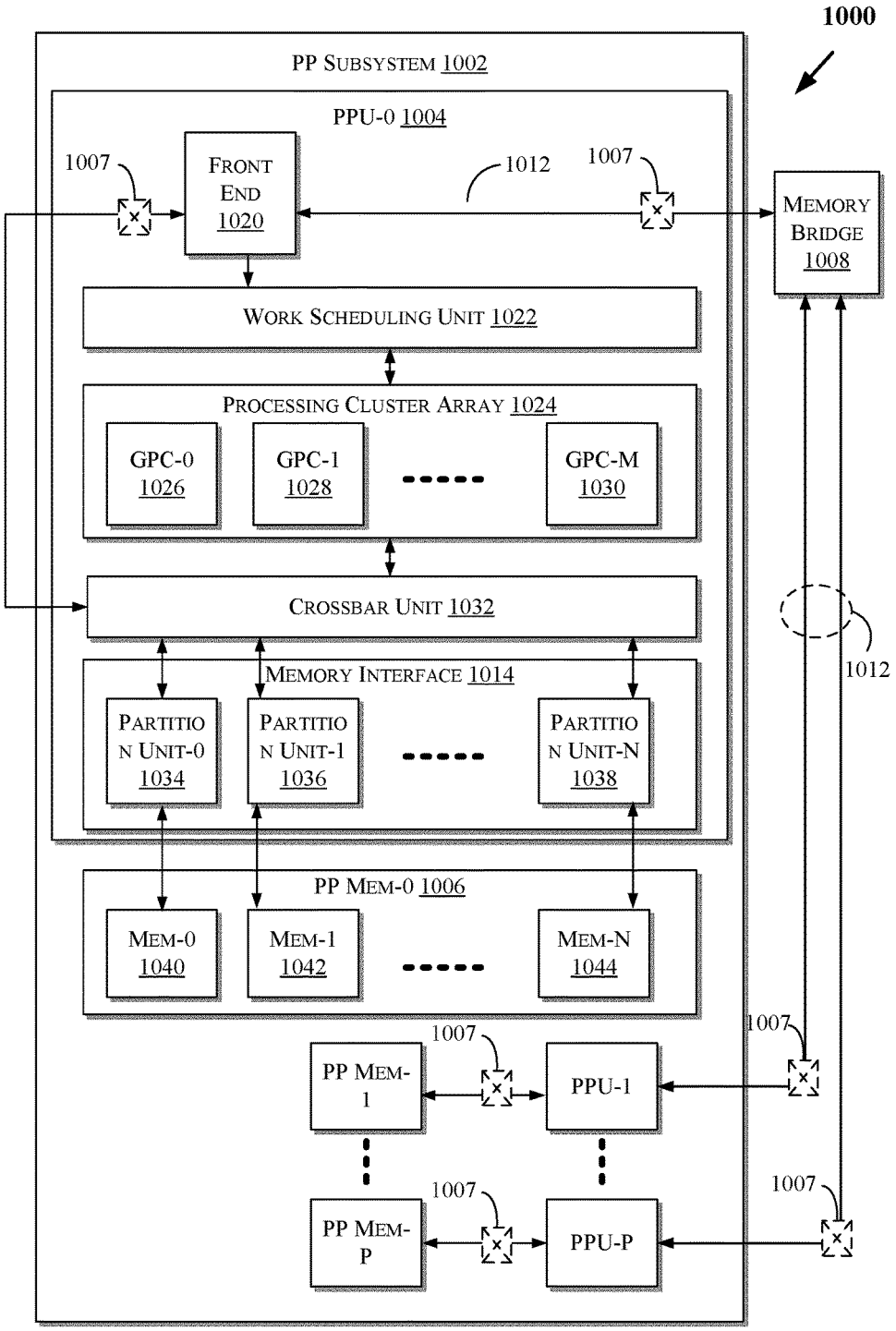

1000
PP SUBSYSTEM 1002
PPU-0 1004
1007
FRONT END 1020
1012
1007
MEMORY BRIDGE 1008
WORK SCHEDULING UNIT 1022
PROCESSING CLUSTER ARRAY 1024
GPC-0 1026
GPC-1 1028
GPC-M 1030
CROSSBAR UNIT 1032
1012
MEMORY INTERFACE 1014
PARTITION UNIT-0 1034
PARTITION UNIT-1 1036
PARTITION UNIT-N 1038
PP MEM-0 1006
MEM-0 1040
MEM-1 1042
MEM-N 1044
1007
1007
PP MEM-1
PPU-1
1007
1007
PP MEM-P
PPU-P

FIG. 10

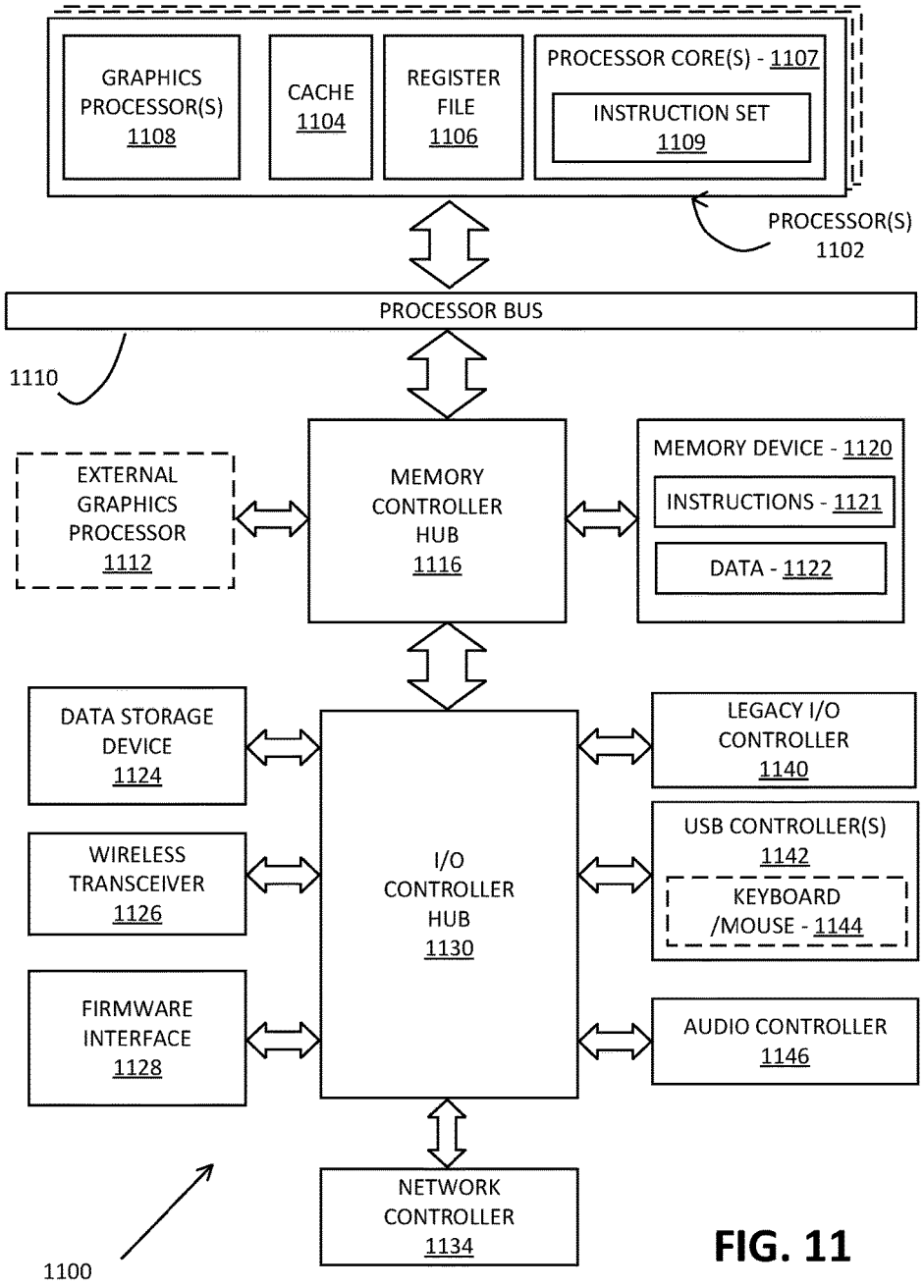

GRAPHICS PROCESSOR(S) 1108
CACHE 1104
REGISTER FILE 1106
PROCESSOR CORE(S) - 1107
INSTRUCTION SET 1109
PROCESSOR(S) 1102
PROCESSOR BUS
1110
EXTERNAL GRAPHICS PROCESSOR 1112
MEMORY CONTROLLER HUB 1116
MEMORY DEVICE - 1120
INSTRUCTIONS - 1121
DATA - 1122
DATA STORAGE DEVICE 1124
WIRELESS TRANSCEIVER 1126
FIRMWARE INTERFACE 1128
I/O CONTROLLER HUB 1130
LEGACY I/O CONTROLLER 1140
USB CONTROLLER(S) 1142
KEYBOARD /MOUSE - 1144
AUDIO CONTROLLER 1146
NETWORK CONTROLLER 1134
1100

FIG. 11

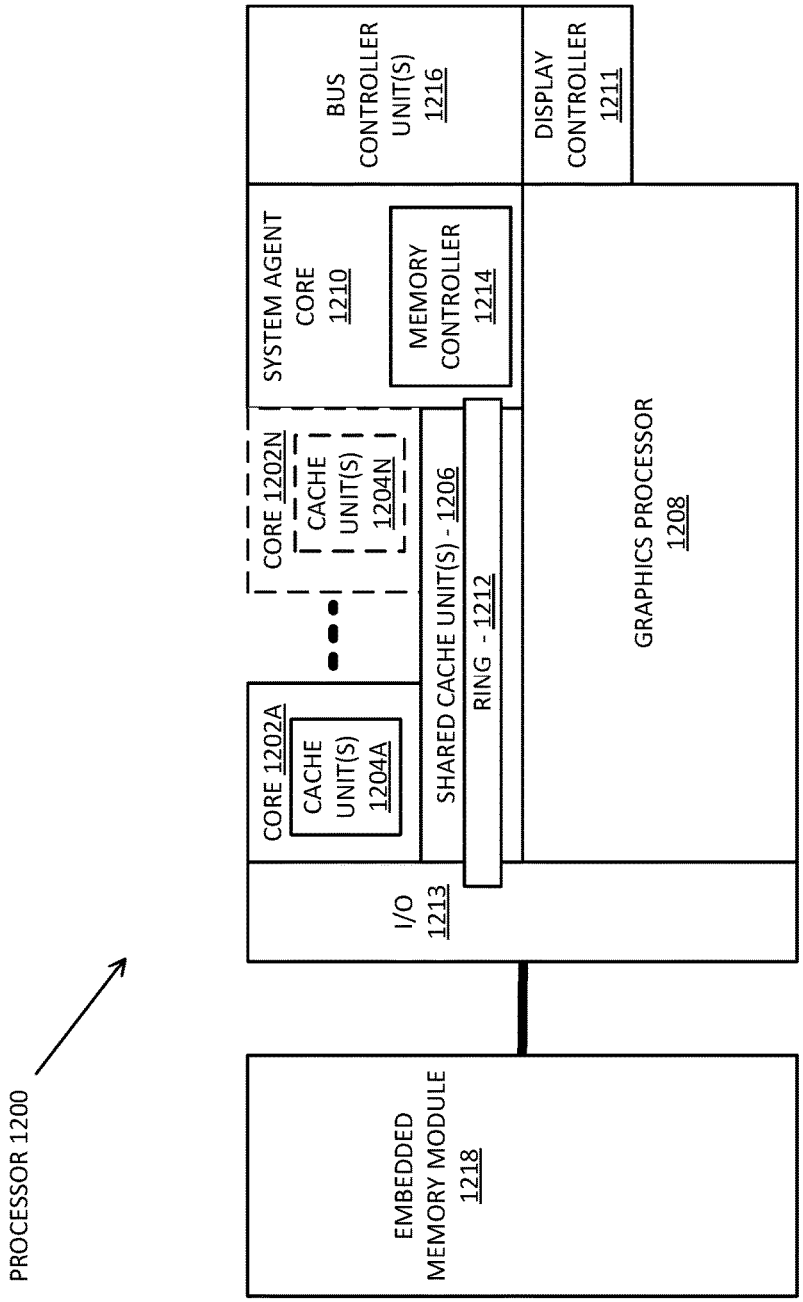

PROCESSOR 1200
CORE 1202A
CACHE UNIT(S) 1204A
CORE 1202N
CACHE UNIT(S) 1204N
SYSTEM AGENT CORE 1210
MEMORY CONTROLLER 1214
BUS CONTROLLER UNIT(S) 1216
DISPLAY CONTROLLER 1211
SHARED CACHE UNIT(S) - 1206
RING - 1212
GRAPHICS PROCESSOR 1208
I/O 1213
EMBEDDED MEMORY MODULE 1218

FIG. 12

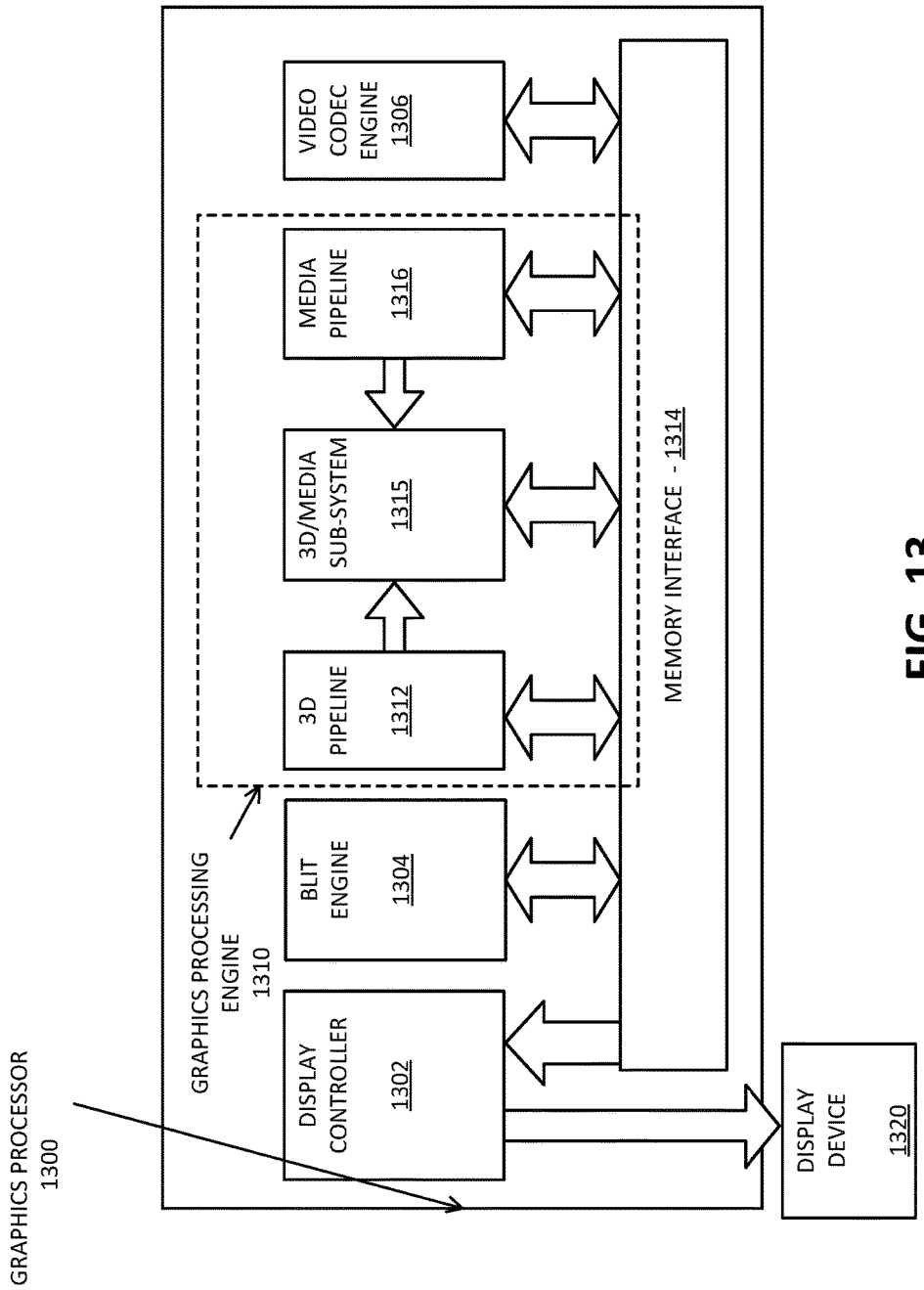

GRAPHICS PROCESSOR
1300
GRAPHICS PROCESSING
ENGINE
1310
DISPLAY
CONTROLLER
1302
BLIT
ENGINE
1304
3D
PIPELINE
1312
3D/MEDIA
SUB-SYSTEM
1315
MEDIA
PIPELINE
1316
VIDEO
CODEC
ENGINE
1306
MEMORY INTERFACE - 1314
DISPLAY
DEVICE
1320

FIG. 13

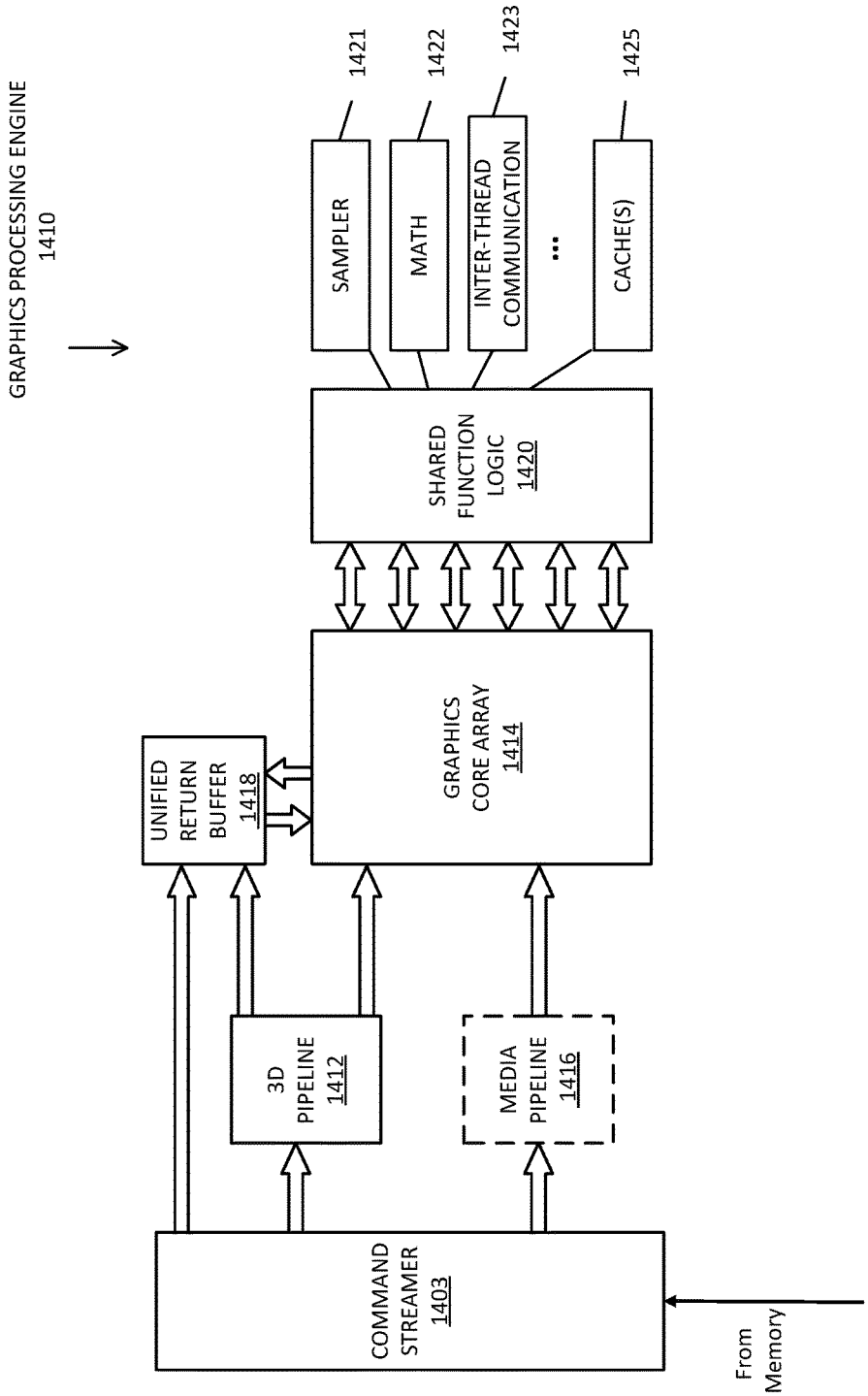

GRAPHICS PROCESSING ENGINE
1410
SAMPLER
1421
MATH
1422
INTER-THREAD COMMUNICATION
1423
...
CACHE(S)
1425
SHARED FUNCTION LOGIC
1420
GRAPHICS CORE ARRAY
1414
UNIFIED RETURN BUFFER
1418
3D PIPELINE
1412
MEDIA PIPELINE
1416
COMMAND STREAMER
1403
From Memory

FIG. 14

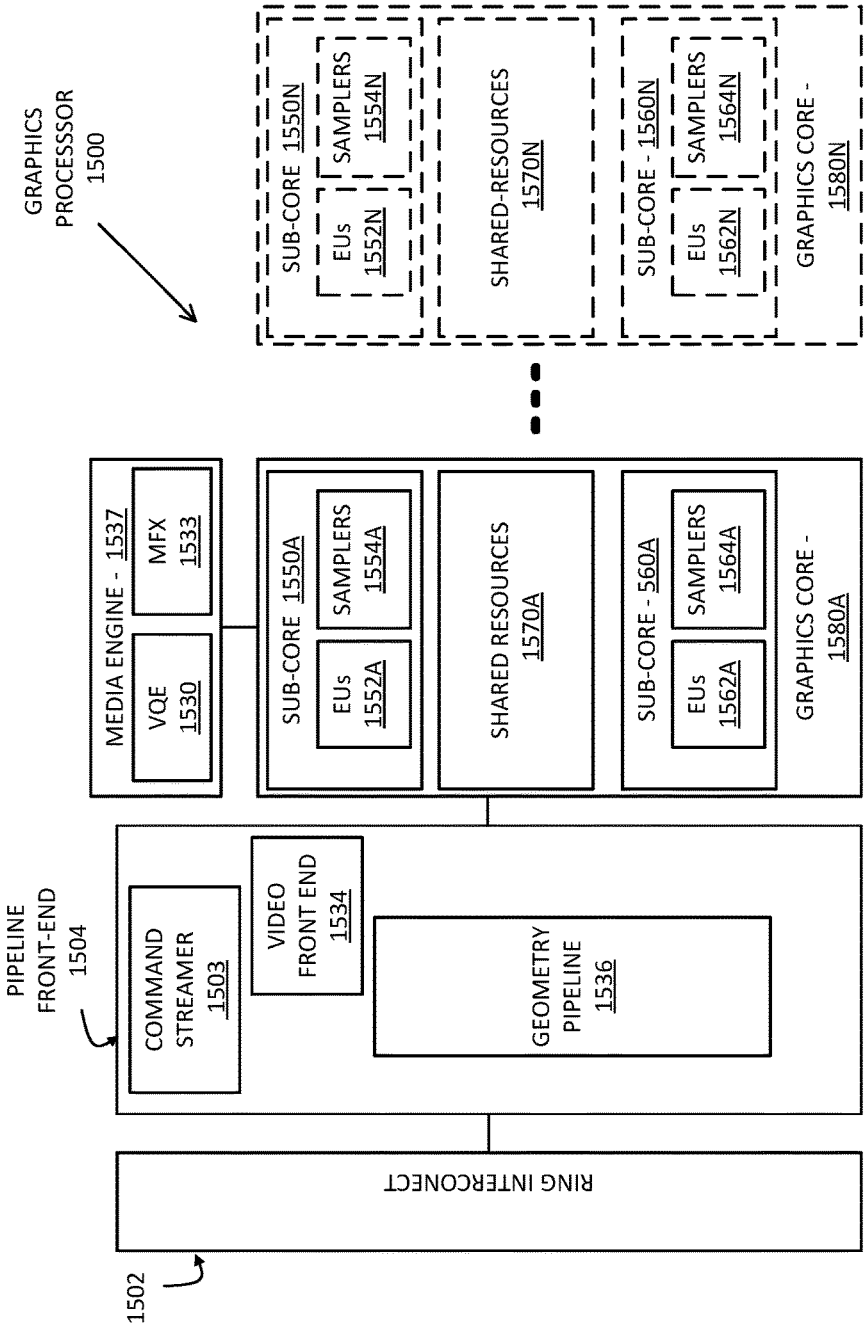

GRAPHICS PROCESSOR 1500
PIPELINE FRONT-END 1504
COMMAND STREAMER 1503
VIDEO FRONT END 1534
GEOMETRY PIPELINE 1536
MEDIA ENGINE - 1537
VQE 1530
MFX 1533
SUB-CORE 1550A
EUs 1552A
SAMPLERS 1554A
SHARED RESOURCES 1570A
SUB-CORE - 560A
EUs 1562A
SAMPLERS 1564A
GRAPHICS CORE - 1580A
SUB-CORE 1550N
EUs 1552N
SAMPLERS 1554N
SHARED-RESOURCES 1570N
SUB-CORE - 1560N
EUs 1562N
SAMPLERS 1564N
GRAPHICS CORE - 1580N
RING INTERCONECT
1502

FIG. 15

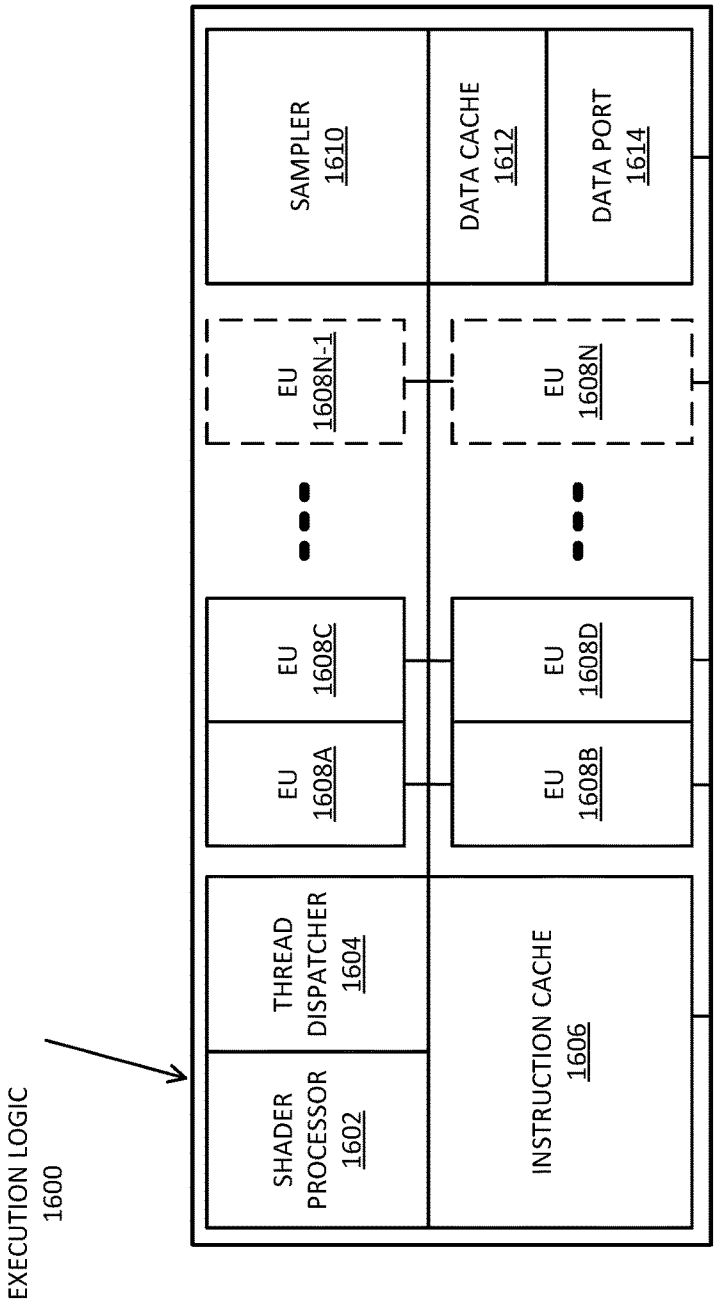

EXECUTION LOGIC
1600
SHADER PROCESSOR
1602
THREAD DISPATCHER
1604
EU
1608A
EU
1608C
EU
1608N-1
INSTRUCTION CACHE
1606
EU
1608B
EU
1608D
EU
1608N
SAMPLER
1610
DATA CACHE
1612
DATA PORT
1614

FIG. 16

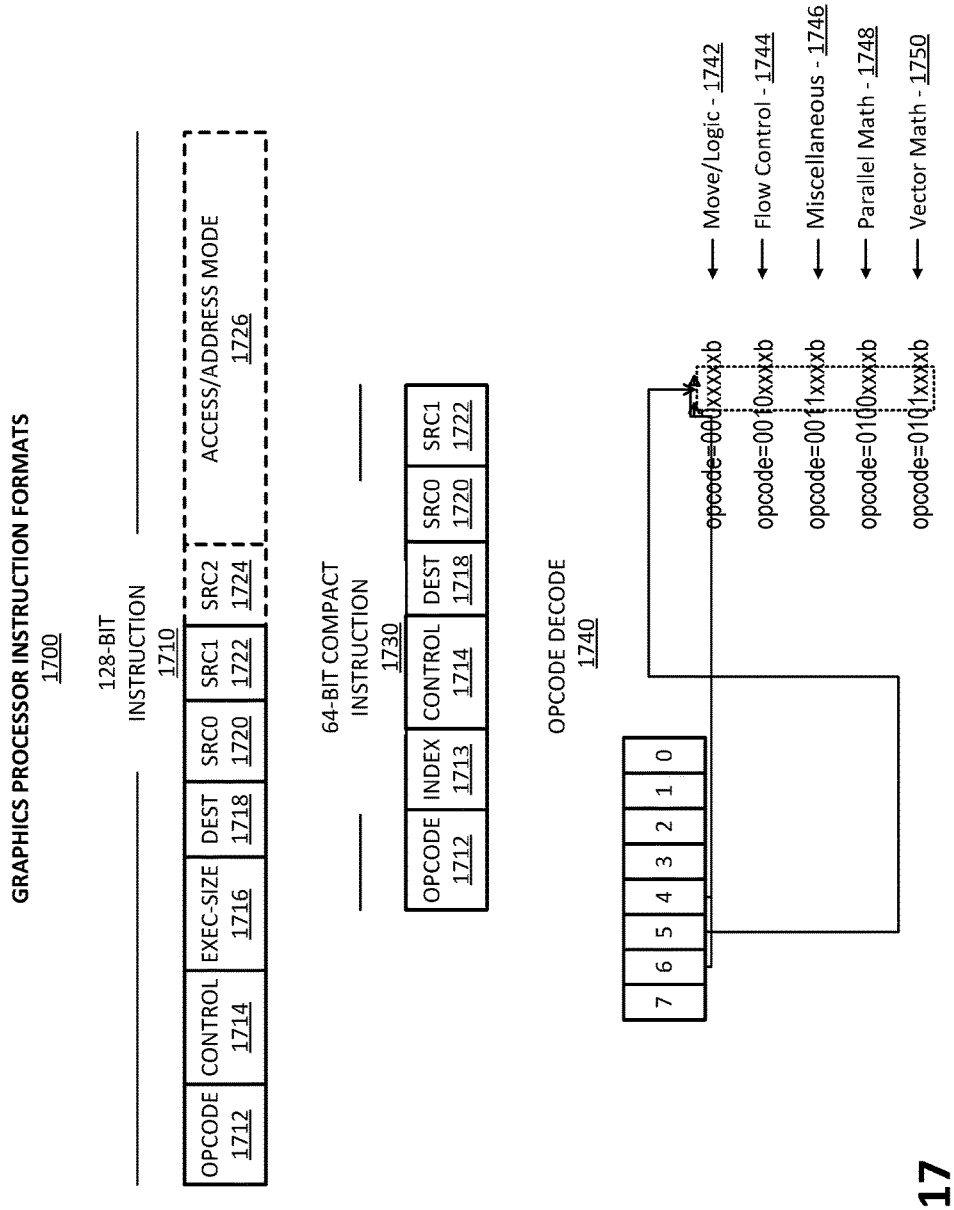

GRAPHICS PROCESSOR INSTRUCTION FORMATS
1700
128-BIT INSTRUCTION
1710
OPCODE 1712
CONTROL 1714
EXEC-SIZE 1716
DEST 1718
SRC0 1720
SRC1 1722
SRC2 1724
ACCESS/ADDRESS MODE 1726
64-BIT COMPACT INSTRUCTION
1730
OPCODE 1712
INDEX 1713
CONTROL 1714
DEST 1718
SRC0 1720
SRC1 1722
OPCODE DECODE
1740
7 6 5 4 3 2 1 0
opcode=000xxxxb
opcode=0010xxxxb
opcode=0011xxxxb
opcode=0100xxxxb
opcode=0101xxxxb
Move/Logic - 1742
Flow Control - 1744
Miscellaneous - 1746
Parallel Math - 1748
Vector Math - 1750

FIG. 17

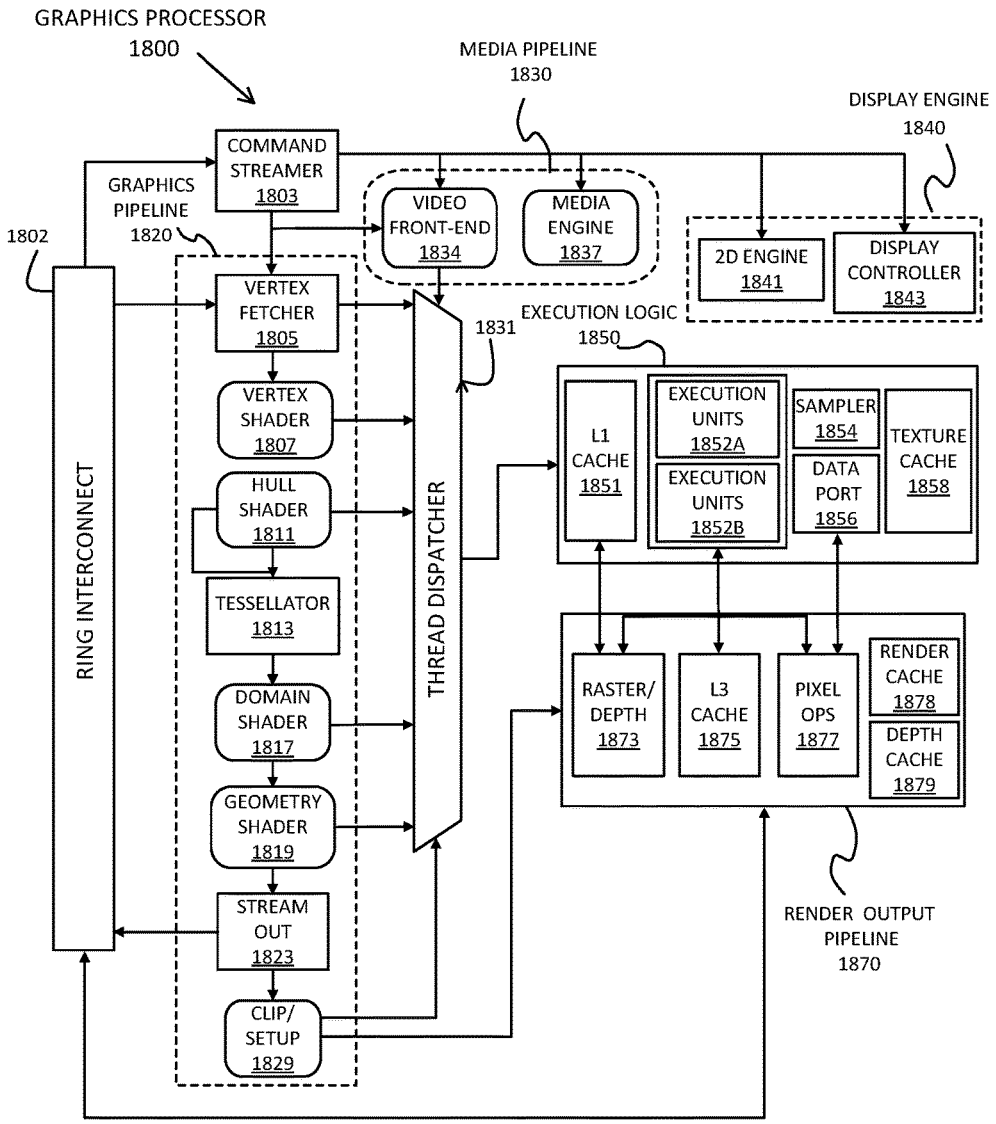

GRAPHICS PROCESSOR
1800
MEDIA PIPELINE
1830
DISPLAY ENGINE
1840
COMMAND STREAMER
1803
GRAPHICS PIPELINE
1820
1802
VIDEO FRONT-END
1834
MEDIA ENGINE
1837
2D ENGINE
1841
DISPLAY CONTROLLER
1843
VERTEX FETCHER
1805
EXECUTION LOGIC
1850
1831
VERTEX SHADER
1807
HULL SHADER
1811
TESSELLATOR
1813
DOMAIN SHADER
1817
GEOMETRY SHADER
1819
STREAM OUT
1823
CLIP/ SETUP
1829
RING INTERCONNECT
THREAD DISPATCHER
L1 CACHE
1851
EXECUTION UNITS
1852A
EXECUTION UNITS
1852B
SAMPLER
1854
DATA PORT
1856
TEXTURE CACHE
1858
RASTER/ DEPTH
1873
L3 CACHE
1875
PIXEL OPS
1877
RENDER CACHE
1878
DEPTH CACHE
1879
RENDER OUTPUT PIPELINE
1870

FIG. 18

FIG. 19A

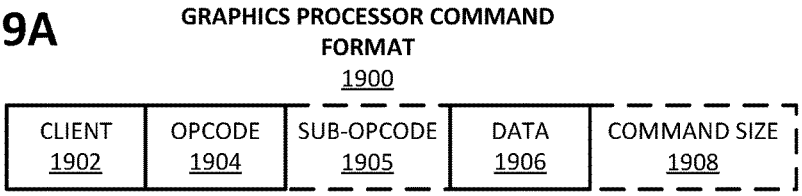
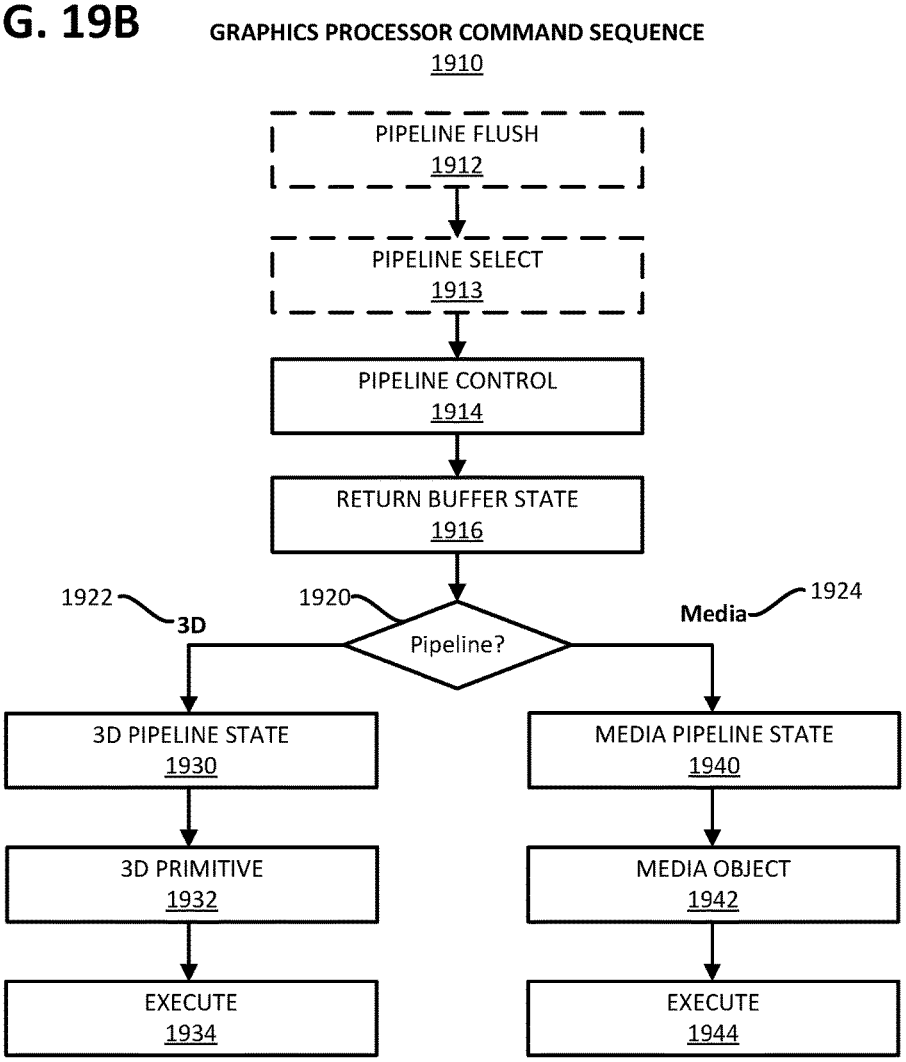


FIG. 19B



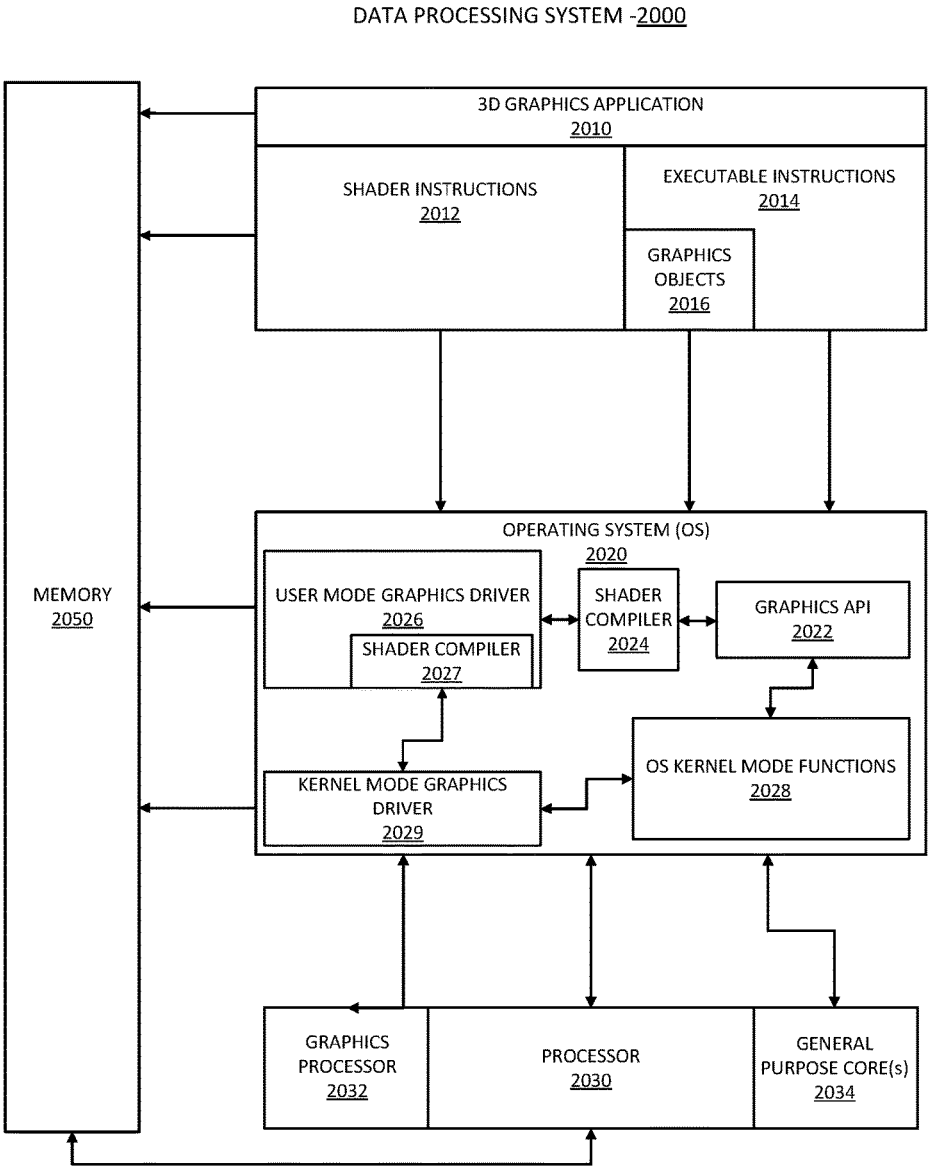

DATA PROCESSING SYSTEM -2000
3D GRAPHICS APPLICATION 2010
SHADER INSTRUCTIONS 2012
EXECUTABLE INSTRUCTIONS 2014
GRAPHICS OBJECTS 2016
MEMORY 2050
OPERATING SYSTEM (OS) 2020
USER MODE GRAPHICS DRIVER 2026
SHADER COMPILER 2027
SHADER COMPILER 2024
GRAPHICS API 2022
KERNEL MODE GRAPHICS DRIVER 2029
OS KERNEL MODE FUNCTIONS 2028
GRAPHICS PROCESSOR 2032
PROCESSOR 2030
GENERAL PURPOSE CORE(s) 2034

FIG. 20

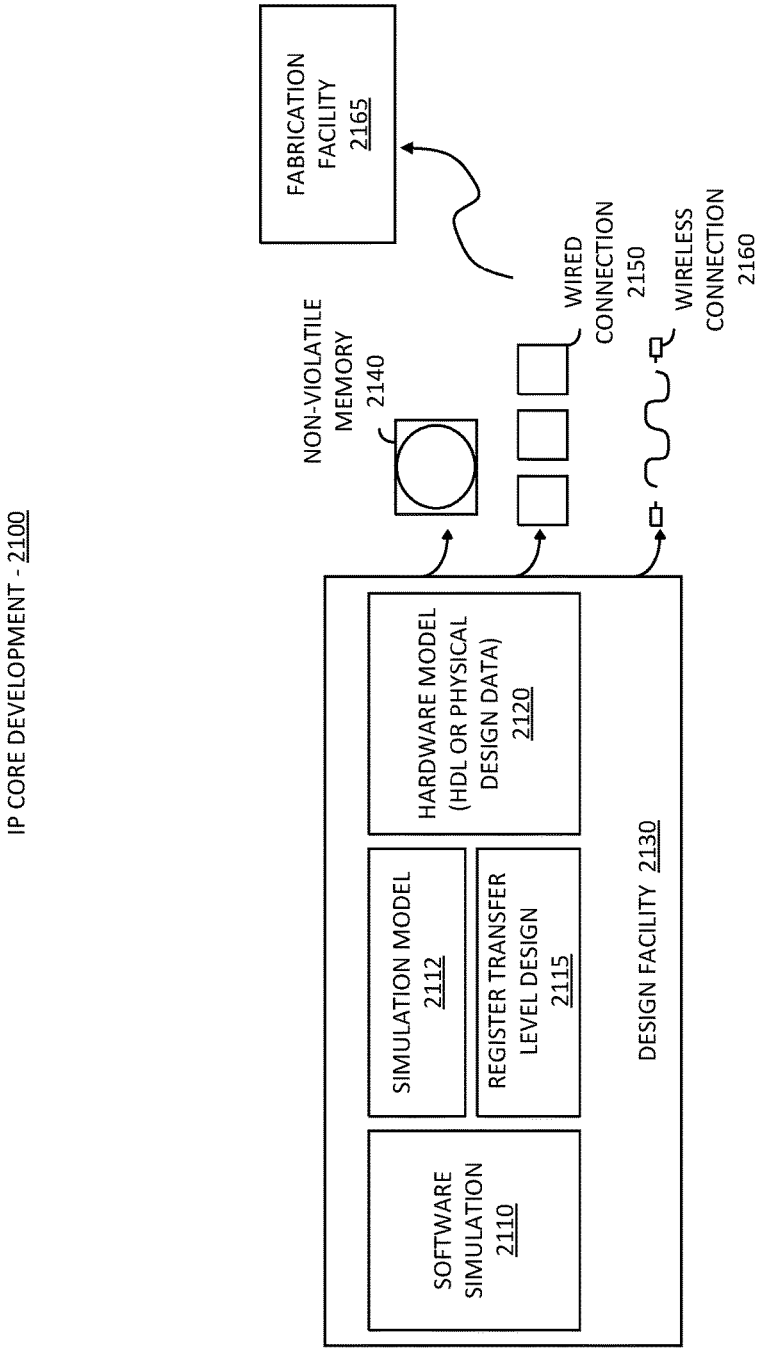

IP CORE DEVELOPMENT - 2100
SOFTWARE SIMULATION 2110
SIMULATION MODEL 2112
REGISTER TRANSFER LEVEL DESIGN 2115
HARDWARE MODEL (HDL OR PHYSICAL DESIGN DATA) 2120
DESIGN FACILITY 2130
NON-VIOLATILE MEMORY 2140
WIRED CONNECTION 2150
WIRELESS CONNECTION 2160
FABRICATION FACILITY 2165

FIG. 21

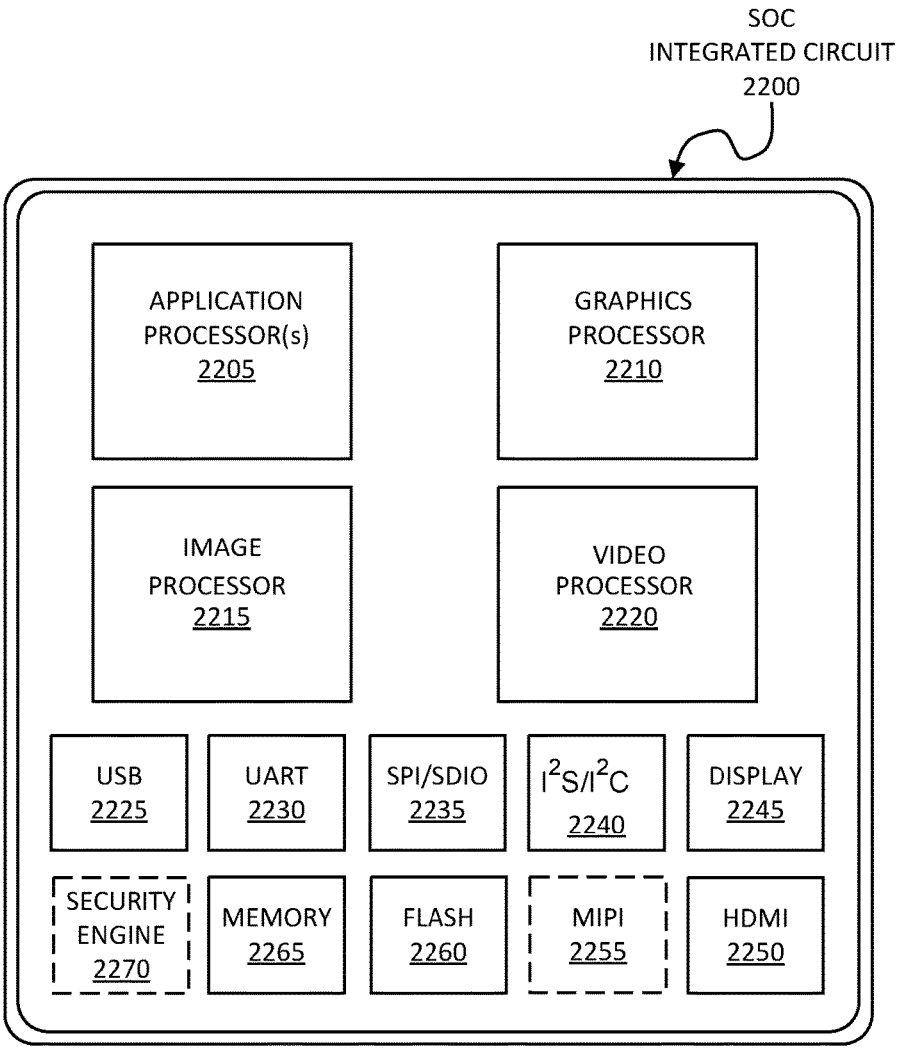

SOC
INTEGRATED CIRCUIT
2200
APPLICATION PROCESSOR(s) 2205
GRAPHICS PROCESSOR 2210
IMAGE PROCESSOR 2215
VIDEO PROCESSOR 2220
USB 2225
UART 2230
SPI/SDIO 2235
I2S/I2C 2240
DISPLAY 2245
SECURITY ENGINE 2270
MEMORY 2265
FLASH 2260
MIPI 2255
HDMI 2250

FIG. 22

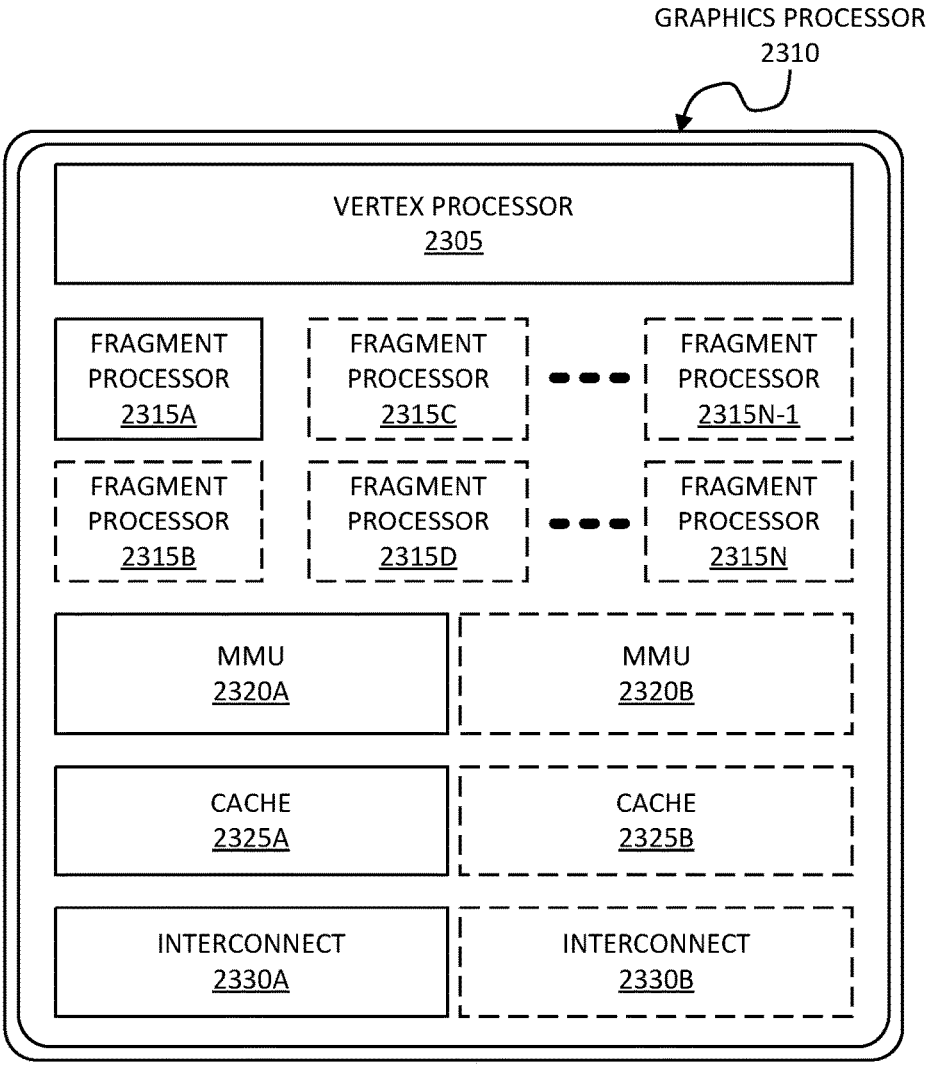

GRAPHICS PROCESSOR
2310
VERTEX PROCESSOR
2305
FRAGMENT PROCESSOR
2315A
FRAGMENT PROCESSOR
2315C
FRAGMENT PROCESSOR
2315N-1
FRAGMENT PROCESSOR
2315B
FRAGMENT PROCESSOR
2315D
FRAGMENT PROCESSOR
2315N
MMU
2320A
MMU
2320B
CACHE
2325A
CACHE
2325B
INTERCONNECT
2330A
INTERCONNECT
2330B

FIG. 23

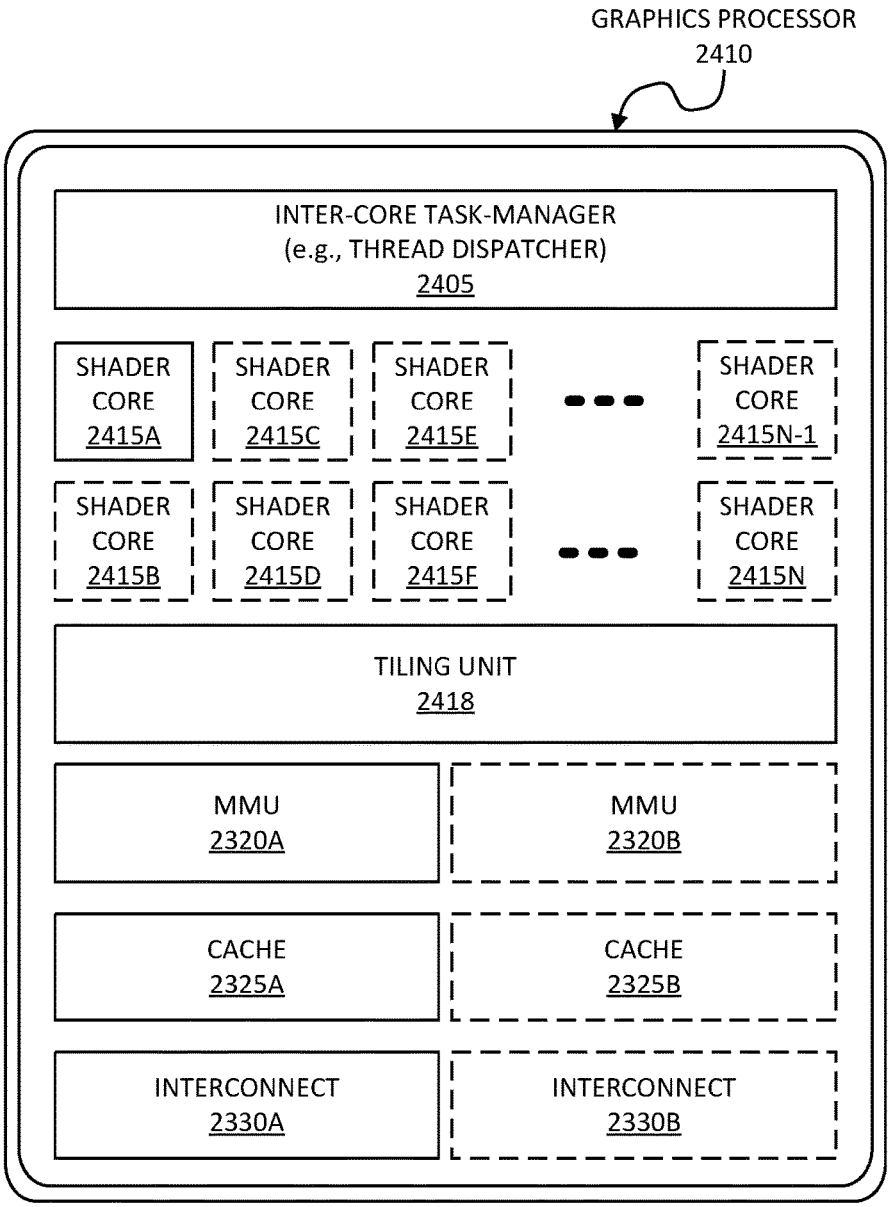

GRAPHICS PROCESSOR
2410
INTER-CORE TASK-MANAGER
(e.g., THREAD DISPATCHER)
2405
SHADER CORE 2415A
SHADER CORE 2415C
SHADER CORE 2415E
SHADER CORE 2415N-1
SHADER CORE 2415B
SHADER CORE 2415D
SHADER CORE 2415F
SHADER CORE 2415N
TILING UNIT
2418
MMU
2320A
MMU
2320B
CACHE
2325A
CACHE
2325B
INTERCONNECT
2330A
INTERCONNECT
2330B

FIG. 24

SYSTEM, APPARATUS AND METHOD FOR SEGMENTING A MEMORY ARRAY

TECHNICAL FIELD

Embodiments relate generally to data processing and more particularly to data processing via a general-purpose graphics processing unit.

BACKGROUND

Current parallel graphics data processing includes systems and methods developed to perform specific operations on graphics data such as, for example, linear interpolation, tessellation, rasterization, texture mapping, depth testing, etc. Traditionally, graphics processors used fixed function computational units to process graphics data; however, more recently, portions of graphics processors have been made programmable, enabling such processors to support a wider variety of operations for processing vertex and fragment data.

To further increase performance, graphics processors typically implement processing techniques such as pipelining that attempt to process, in parallel, as much graphics data as possible throughout the different parts of the graphics pipeline. Parallel graphics processors with single instruction, multiple thread (SIMT) architectures are designed to maximize the amount of parallel processing in the graphics pipeline. In an SIMT architecture, groups of parallel threads attempt to execute program instructions synchronously together as often as possible to increase processing efficiency. A general overview of software and hardware for SIMT architectures can be found in Shane Cook, *CUDA Programming*, Chapter 3, pages 37-51 (2013) and/or Nicholas Wilt, CUDA Handbook, *A Comprehensive Guide to GPU Programming*, Sections 2.6.2 to 3.1.2 (June 2013).

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram illustrating a computer system configured to implement one or more aspects of the embodiments described herein.

FIG. **2**A-**2**D illustrate a parallel processor components, according to an embodiment.

FIGS. **3**A-**3**B are block diagrams of graphics multiprocessors, according to embodiments.

FIG. **4**A-**4**F illustrate an exemplary architecture in which a plurality of GPUs are communicatively coupled to a plurality of multi-core processors.

FIG. **5** illustrates a graphics processing pipeline, according to an embodiment.

FIG. **6** is a schematic diagram of a memory circuit in accordance with an embodiment of the present invention.

FIG. **7** is a flow diagram of a method in accordance with an embodiment of the present invention.

FIG. **8** illustrates a block diagram of a switching regulator according to an embodiment.

FIG. **9** is a block diagram of a system including a streaming multiprocessor in accordance with one or more embodiments.

FIG. **10** illustrates a block diagram of a parallel processing system according to one embodiment.

FIG. **11** is a block diagram of a processing system according to an embodiment.

FIG. **12** is a block diagram of an embodiment of a processor having one or more processor cores, an integrated memory controller, and an integrated graphics processor.

FIG. **13** is a block diagram of a graphics processor, which may be a discrete graphics processing unit, or may be a graphics processor integrated with a plurality of processing cores.

FIG. **14** is a block diagram of a graphics processing engine of a graphics processor in accordance with some embodiments.

FIG. **15** is a block diagram of another embodiment of a graphics processor.

FIG. **16** illustrates thread execution logic including an array of processing elements employed in some embodiments of a GPE.

FIG. **17** is a block diagram illustrating a graphics processor instruction formats according to some embodiments.

FIG. **18** is a block diagram of another embodiment of a graphics processor.

FIG. **19**A is a block diagram illustrating a graphics processor command format according to some embodiments.

FIG. **19**B is a block diagram illustrating a graphics processor command sequence according to an embodiment.

FIG. **20** illustrates exemplary graphics software architecture for a data processing system according to some embodiments.

FIG. **21** is a block diagram illustrating an IP core development system that may be used to manufacture an integrated circuit to perform operations according to an embodiment.

FIG. **22** is a block diagram illustrating an exemplary system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment.

FIG. **23** is a block diagram illustrating an exemplary graphics processor of a system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment.

FIG. **24** is a block diagram illustrating an additional exemplary graphics processor of a system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment.

DETAILED DESCRIPTION

In some embodiments, a graphics processing unit (GPU) is communicatively coupled to host/processor cores to accelerate graphics operations, machine-learning operations, pattern analysis operations, and various general purpose GPU (GPGPU) functions. The GPU may be communicatively coupled to the host processor/cores over a bus or another interconnect (e.g., a high-speed interconnect such as PCIe or NVLink). In other embodiments, the GPU may be integrated on the same package or chip as the cores and communicatively coupled to the cores over an internal processor bus/interconnect (i.e., internal to the package or chip). Regardless of the manner in which the GPU is connected, the processor cores may allocate work to the GPU in the form of sequences of commands/instructions contained in a work descriptor. The GPU then uses dedicated circuitry/logic for efficiently processing these commands/instructions.

In the following description, numerous specific details are set forth to provide a more thorough understanding. However, it will be apparent to one of skill in the art that the embodiments described herein may be practiced without one or more of these specific details. In other instances, well-known features have not been described to avoid obscuring the details of the present embodiments.

System Overview

FIG. **1** is a block diagram illustrating a computing system **100** configured to implement one or more aspects of the embodiments described herein. The computing system **100** includes a processing subsystem **101** having one or more processor(s) **102** and a system memory **104** communicating via an interconnection path that may include a memory hub **105**. The memory hub **105** may be a separate component within a chipset component or may be integrated within the one or more processor(s) **102**. The memory hub **105** couples with an I/O subsystem **111** via a communication link **106**. The I/O subsystem **111** includes an I/O hub **107** that can enable the computing system **100** to receive input from one or more input device(s) **108**. Additionally, the I/O hub **107** can enable a display controller, which may be included in the one or more processor(s) **102**, to provide outputs to one or more display device(s) **110**A. In one embodiment the one or more display device(s) **110**A coupled with the I/O hub **107** can include a local, internal, or embedded display device.

In one embodiment the processing subsystem **101** includes one or more parallel processor(s) **112** coupled to memory hub **105** via a bus or other communication link **113**. The communication link **113** may be one of any number of standards based communication link technologies or protocols, such as, but not limited to PCI Express, or may be a vendor specific communications interface or communications fabric. In one embodiment the one or more parallel processor(s) **112** form a computationally focused parallel or vector processing system that an include a large number of processing cores and/or processing clusters, such as a many integrated core (MIC) processor. In one embodiment the one or more parallel processor(s) **112** form a graphics processing subsystem that can output pixels to one of the one or more display device(s) **110**A coupled via the I/O Hub **107**. The one or more parallel processor(s) **112** can also include a display controller and display interface (not shown) to enable a direct connection to one or more display device(s) **110**B.

Within the I/O subsystem **111**, a system storage unit **114** can connect to the I/O hub **107** to provide a storage mechanism for the computing system **100**. An I/O switch **116** can be used to provide an interface mechanism to enable connections between the I/O hub **107** and other components, such as a network adapter **118** and/or wireless network adapter **119** that may be integrated into the platform, and various other devices that can be added via one or more add-in device(s) **120**. The network adapter **118** can be an Ethernet adapter or another wired network adapter. The wireless network adapter **119** can include one or more of a Wi-Fi, Bluetooth, near field communication (NFC), or other network device that includes one or more wireless radios.

The computing system **100** can include other components not explicitly shown, including USB or other port connections, optical storage drives, video capture devices, and the like, may also be connected to the I/O hub **107**. Communication paths interconnecting the various components in FIG. **1** may be implemented using any suitable protocols, such as PCI (Peripheral Component Interconnect) based protocols (e.g., PCI-Express), or any other bus or point-to-point communication interfaces and/or protocol(s), such as the NV-Link high-speed interconnect, or interconnect protocols known in the art.

In one embodiment, the one or more parallel processor(s) **112** incorporate circuitry optimized for graphics and video processing, including, for example, video output circuitry, and constitutes a graphics processing unit (GPU). In another embodiment, the one or more parallel processor(s) **112** incorporate circuitry optimized for general purpose processing, while preserving the underlying computational architecture, described in greater detail herein. In yet another embodiment, components of the computing system **100** may be integrated with one or more other system elements on a single integrated circuit. For example, the one or more parallel processor(s), **112** memory hub **105**, processor(s) **102**, and I/O hub **107** can be integrated into a system on chip (SoC) integrated circuit. Alternatively, the components of the computing system **100** can be integrated into a single package to form a system in package (SIP) configuration. In one embodiment at least a portion of the components of the computing system **100** can be integrated into a multi-chip module (MCM), which can be interconnected with other multi-chip modules into a modular computing system.

It will be appreciated that the computing system **100** shown herein is illustrative and that variations and modifications are possible. The connection topology, including the number and arrangement of bridges, the number of processor(s) **102**, and the number of parallel processor(s) **112**, may be modified as desired. For instance, in some embodiments, system memory **104** is connected to the processor(s) **102** directly rather than through a bridge, while other devices communicate with system memory **104** via the memory hub **105** and the processor(s) **102**. In other alternative topologies, the parallel processor(s) **112** are connected to the I/O hub **107** or directly to one of the one or more processor(s) **102**, rather than to the memory hub **105**. In other embodiments, the I/O hub **107** and memory hub **105** may be integrated into a single chip. Some embodiments may include two or more sets of processor(s) **102** attached via multiple sockets, which can couple with two or more instances of the parallel processor(s) **112**.

Some of the particular components shown herein are optional and may not be included in all implementations of the computing system **100**. For example, any number of add-in cards or peripherals may be supported, or some components may be eliminated. Furthermore, some architectures may use different terminology for components similar to those illustrated in FIG. **1**. For example, the memory hub **105** may be referred to as a Northbridge in some architectures, while the I/O hub **107** may be referred to as a Southbridge.

FIG. **2**A illustrates a parallel processor **200**, according to an embodiment. The various components of the parallel processor **200** may be implemented using one or more integrated circuit devices, such as programmable processors, application specific integrated circuits (ASICs), or field programmable gate arrays (FPGA). The illustrated parallel processor **200** is a variant of the one or more parallel processor(s) **112** shown in FIG. **1**, according to an embodiment.

In one embodiment the parallel processor **200** includes a parallel processing unit **202**. The parallel processing unit includes an I/O unit **204** that enables communication with other devices, including other instances of the parallel processing unit **202**. The I/O unit **204** may be directly connected to other devices. In one embodiment the I/O unit **204** connects with other devices via the use of a hub or switch interface, such as memory hub **105**. The connections between the memory hub **105** and the I/O unit **204** form a communication link **113**. Within the parallel processing unit **202**, the I/O unit **204** connects with a host interface **206** and a memory crossbar **216**, where the host interface **206** receives commands directed to performing processing operations and the memory crossbar **216** receives commands directed to performing memory operations.

When the host interface **206** receives a command buffer via the I/O unit **204**, the host interface **206** can direct work operations to perform those commands to a front end **208**. In one embodiment the front end **208** couples with a scheduler **210**, which is configured to distribute commands or other work items to a processing cluster array **212**. In one embodiment the scheduler **210** ensures that the processing cluster array **212** is properly configured and in a valid state before tasks are distributed to the processing clusters of the processing cluster array **212**. In one embodiment the scheduler **210** is implemented via firmware logic executing on a microcontroller. The microcontroller implemented scheduler **210** is configurable to perform complex scheduling and work distribution operations at coarse and fine granularity, enabling rapid preemption and context switching of threads executing on the processing array **212**. In one embodiment, the host software can prove workloads for scheduling on the processing array **212** via one of multiple graphics processing doorbells. The workloads can then be automatically distributed across the processing array **212** by the scheduler **210** logic within the scheduler microcontroller.

The processing cluster array **212** can include up to "N" processing clusters (e.g., cluster **214**A, cluster **214**B, through cluster **214**N). Each cluster **214**A-**214**N of the processing cluster array **212** can execute a large number of concurrent threads. The scheduler **210** can allocate work to the clusters **214**A-**214**N of the processing cluster array **212** using various scheduling and/or work distribution algorithms, which may vary depending on the workload arising for each type of program or computation. The scheduling can be handled dynamically by the scheduler **210**, or can be assisted in part by compiler logic during compilation of program logic configured for execution by the processing cluster array **212**. In one embodiment, different clusters **214**A-**214**N of the processing cluster array **212** can be allocated for processing different types of programs or for performing different types of computations.

The processing cluster array **212** can be configured to perform various types of parallel processing operations. In one embodiment the processing cluster array **212** is configured to perform general-purpose parallel compute operations. For example, the processing cluster array **212** can include logic to execute processing tasks including filtering of video and/or audio data, performing modeling operations, including physics operations, and performing data transformations.

In one embodiment the processing cluster array **212** is configured to perform parallel graphics processing operations. In embodiments in which the parallel processor **200** is configured to perform graphics processing operations, the processing cluster array **212** can include additional logic to support the execution of such graphics processing operations, including, but not limited to texture sampling logic to perform texture operations, as well as tessellation logic and other vertex processing logic. Additionally, the processing cluster array **212** can be configured to execute graphics processing related shader programs such as, but not limited to vertex shaders, tessellation shaders, geometry shaders, and pixel shaders. The parallel processing unit **202** can transfer data from system memory via the I/O unit **204** for processing. During processing the transferred data can be stored to on-chip memory (e.g., parallel processor memory **222**) during processing, then written back to system memory.

In one embodiment, when the parallel processing unit **202** is used to perform graphics processing, the scheduler **210** can be configured to divide the processing workload into approximately equal sized tasks, to better enable distribution of the graphics processing operations to multiple clusters **214**A-**214**N of the processing cluster array **212**. In some embodiments, portions of the processing cluster array **212** can be configured to perform different types of processing. For example a first portion may be configured to perform vertex shading and topology generation, a second portion may be configured to perform tessellation and geometry shading, and a third portion may be configured to perform pixel shading or other screen space operations, to produce a rendered image for display. Intermediate data produced by one or more of the clusters **214**A-**214**N may be stored in buffers to allow the intermediate data to be transmitted between clusters **214**A-**214**N for further processing.

During operation, the processing cluster array **212** can receive processing tasks to be executed via the scheduler **210**, which receives commands defining processing tasks from front end **208**. For graphics processing operations, processing tasks can include indices of data to be processed, e.g., surface (patch) data, primitive data, vertex data, and/or pixel data, as well as state parameters and commands defining how the data is to be processed (e.g., what program is to be executed). The scheduler **210** may be configured to fetch the indices corresponding to the tasks or may receive the indices from the front end **208**. The front end **208** can be configured to ensure the processing cluster array **212** is configured to a valid state before the workload specified by incoming command buffers (e.g., batch-buffers, push buffers, etc.) is initiated.

Each of the one or more instances of the parallel processing unit **202** can couple with parallel processor memory **222**. The parallel processor memory **222** can be accessed via the memory crossbar **216**, which can receive memory requests from the processing cluster array **212** as well as the I/O unit **204**. The memory crossbar **216** can access the parallel processor memory **222** via a memory interface **218**. The memory interface **218** can include multiple partition units (e.g., partition unit **220**A, partition unit **220**B, through partition unit **220**N) that can each couple to a portion (e.g., memory unit) of parallel processor memory **222**. In one implementation the number of partition units **220**A-**220**N is configured to be equal to the number of memory units, such that a first partition unit **220**A has a corresponding first memory unit **224**A, a second partition unit **220**B has a corresponding memory unit **224**B, and an Nth partition unit **220**N has a corresponding Nth memory unit **224**N. In other embodiments, the number of partition units **220**A-**220**N may not be equal to the number of memory devices.

In various embodiments, the memory units **224**A-**224**N can include various types of memory devices, including dynamic random access memory (DRAM) or graphics random access memory, such as synchronous graphics random access memory (SGRAM), including graphics double data rate (GDDR) memory. In one embodiment, the memory units **224**A-**224**N may also include 3D stacked memory, including but not limited to high bandwidth memory (HBM). Persons skilled in the art will appreciate that the specific implementation of the memory units **224**A-**224**N can vary, and can be selected from one of various conventional designs. Render targets, such as frame buffers or texture maps may be stored across the memory units **224**A-**224**N, allowing partition units **220**A-**220**N to write portions of each render target in parallel to efficiently use the available bandwidth of parallel processor memory **222**. In some embodiments, a local instance of the parallel processor

memory **222** may be excluded in favor of a unified memory design that utilizes system memory in conjunction with local cache memory.

In one embodiment, any one of the clusters **214**A-**214**N of the processing cluster array **212** can process data that will be written to any of the memory units **224**A-**224**N within parallel processor memory **222**. The memory crossbar **216** can be configured to transfer the output of each cluster **214**A-**214**N to any partition unit **220**A-**220**N or to another cluster **214**A-**214**N, which can perform additional processing operations on the output. Each cluster **214**A-**214**N can communicate with the memory interface **218** through the memory crossbar **216** to read from or write to various external memory devices. In one embodiment the memory crossbar **216** has a connection to the memory interface **218** to communicate with the I/O unit **204**, as well as a connection to a local instance of the parallel processor memory **222**, enabling the processing units within the different processing clusters **214**A-**214**N to communicate with system memory or other memory that is not local to the parallel processing unit **202**. In one embodiment the memory crossbar **216** can use virtual channels to separate traffic streams between the clusters **214**A-**214**N and the partition units **220**A-**220**N.

While a single instance of the parallel processing unit **202** is illustrated within the parallel processor **200**, any number of instances of the parallel processing unit **202** can be included. For example, multiple instances of the parallel processing unit **202** can be provided on a single add-in card, or multiple add-in cards can be interconnected. The different instances of the parallel processing unit **202** can be configured to inter-operate even if the different instances have different numbers of processing cores, different amounts of local parallel processor memory, and/or other configuration differences. For example and in one embodiment, some instances of the parallel processing unit **202** can include higher precision floating point units relative to other instances. Systems incorporating one or more instances of the parallel processing unit **202** or the parallel processor **200** can be implemented in a variety of configurations and form factors, including but not limited to desktop, laptop, or handheld personal computers, servers, workstations, game consoles, and/or embedded systems.

FIG. **2**B is a block diagram of a partition unit **220**, according to an embodiment. In one embodiment the partition unit **220** is an instance of one of the partition units **220**A-**220**N of FIG. **2**A. As illustrated, the partition unit **220** includes an L2 cache **221**, a frame buffer interface **225**, and a ROP **226** (raster operations unit). The L2 cache **221** is a read/write cache that is configured to perform load and store operations received from the memory crossbar **216** and ROP **226**. Read misses and urgent write-back requests are output by L2 cache **221** to frame buffer interface **225** for processing. Updates can also be sent to the frame buffer via the frame buffer interface **225** for processing. In one embodiment the frame buffer interface **225** interfaces with one of the memory units in parallel processor memory, such as the memory units **224**A-**224**N of FIG. **2** (e.g., within parallel processor memory **222**).

In graphics applications, the ROP **226** is a processing unit that performs raster operations such as stencil, z test, blending, and the like. The ROP **226** then outputs processed graphics data that is stored in graphics memory. In some embodiments the ROP **226** includes compression logic to compress depth or color data that is written to memory and decompress depth or color data that is read from memory. The compression logic can be lossless compression logic that makes use of one or more of multiple compression algorithms. The type of compression that is performed by the ROP **226** can vary based on the statistical characteristics of the data to be compressed. For example, in one embodiment, delta color compression is performed on depth and color data on a per-tile basis.

In some embodiments, the ROP **226** is included within each processing cluster (e.g., cluster **214**A-**214**N of FIG. **2**) instead of within the partition unit **220**. In such embodiment, read and write requests for pixel data are transmitted over the memory crossbar **216** instead of pixel fragment data. The processed graphics data may be displayed on a display device, such as one of the one or more display device(s) **110** of FIG. **1**, routed for further processing by the processor(s) **102**, or routed for further processing by one of the processing entities within the parallel processor **200** of FIG. **2**A.

FIG. **2**C is a block diagram of a processing cluster **214** within a parallel processing unit, according to an embodiment. In one embodiment the processing cluster is an instance of one of the processing clusters **214**A-**214**N of FIG. **2**. The processing cluster **214** can be configured to execute many threads in parallel, where the term "thread" refers to an instance of a particular program executing on a particular set of input data. In some embodiments, single-instruction, multiple-data (SIMD) instruction issue techniques are used to support parallel execution of a large number of threads without providing multiple independent instruction units. In other embodiments, single-instruction, multiple-thread (SIMT) techniques are used to support parallel execution of a large number of generally synchronized threads, using a common instruction unit configured to issue instructions to a set of processing engines within each one of the processing clusters. Unlike a SIMD execution regime, where all processing engines typically execute identical instructions, SIMT execution allows different threads to more readily follow divergent execution paths through a given thread program. Persons skilled in the art will understand that a SIMD processing regime represents a functional subset of a SIMT processing regime.

Operation of the processing cluster **214** can be controlled via a pipeline manager **232** that distributes processing tasks to SIMT parallel processors. The pipeline manager **232** receives instructions from the scheduler **210** of FIG. **2** and manages execution of those instructions via a graphics multiprocessor **234** and/or a texture unit **236**. The illustrated graphics multiprocessor **234** is an exemplary instance of a SIMT parallel processor. However, various types of SIMT parallel processors of differing architectures may be included within the processing cluster **214**. One or more instances of the graphics multiprocessor **234** can be included within a processing cluster **214**. The graphics multiprocessor **234** can process data and a data crossbar **240** can be used to distribute the processed data to one of multiple possible destinations, including other shader units. The pipeline manager **232** can facilitate the distribution of processed data by specifying destinations for processed data to be distributed vis the data crossbar **240**.

Each graphics multiprocessor **234** within the processing cluster **214** can include an identical set of functional execution logic (e.g., arithmetic logic units, load-store units, etc.). The functional execution logic can be configured in a pipelined manner in which new instructions can be issued before previous instructions are complete. The functional execution logic supports a variety of operations including integer and floating point arithmetic, comparison operations, Boolean operations, bit-shifting, and computation of various algebraic functions. In one embodiment the same functional-

unit hardware can be leveraged to perform different operations and any combination of functional units may be present.

The instructions transmitted to the processing cluster **214** constitutes a thread. A set of threads executing across the set of parallel processing engines is a thread group. A thread group executes the same program on different input data. Each thread within a thread group can be assigned to a different processing engine within a graphics multiprocessor **234**. A thread group may include fewer threads than the number of processing engines within the graphics multiprocessor **234**. When a thread group includes fewer threads than the number of processing engines, one or more of the processing engines may be idle during cycles in which that thread group is being processed. A thread group may also include more threads than the number of processing engines within the graphics multiprocessor **234**. When the thread group includes more threads than the number of processing engines within the graphics multiprocessor **234**, processing can be performed over consecutive clock cycles. In one embodiment multiple thread groups can be executed concurrently on a graphics multiprocessor **234**.

In one embodiment the graphics multiprocessor **234** includes an internal cache memory to perform load and store operations. In one embodiment, the graphics multiprocessor **234** can forego an internal cache and use a cache memory (e.g., L1 cache **308**) within the processing cluster **214**. Each graphics multiprocessor **234** also has access to L2 caches within the partition units (e.g., partition units **220**A-**220**N of FIG. **2**) that are shared among all processing clusters **214** and may be used to transfer data between threads. The graphics multiprocessor **234** may also access off-chip global memory, which can include one or more of local parallel processor memory and/or system memory. Any memory external to the parallel processing unit **202** may be used as global memory. Embodiments in which the processing cluster **214** includes multiple instances of the graphics multiprocessor **234** can share common instructions and data, which may be stored in the L1 cache **308**.

Each processing cluster **214** may include an MMU **245** (memory management unit) that is configured to map virtual addresses into physical addresses. In other embodiments, one or more instances of the MMU **245** may reside within the memory interface **218** of FIG. **2**. The MMU **245** includes a set of page table entries (PTEs) used to map a virtual address to a physical address of a tile (talk more about tiling) and optionally a cache line index. The MMU **245** may include address translation lookaside buffers (TLB) or caches that may reside within the graphics multiprocessor **234** or the L1 cache or processing cluster **214**. The physical address is processed to distribute surface data access locality to allow efficient request interleaving among partition units. The cache line index may be used to determine whether a request for a cache line is a hit or miss.

In graphics and computing applications, a processing cluster **214** may be configured such that each graphics multiprocessor **234** is coupled to a texture unit **236** for performing texture mapping operations, e.g., determining texture sample positions, reading texture data, and filtering the texture data. Texture data is read from an internal texture L1 cache (not shown) or in some embodiments from the L1 cache within graphics multiprocessor **234** and is fetched from an L2 cache, local parallel processor memory, or system memory, as needed. Each graphics multiprocessor **234** outputs processed tasks to the data crossbar **240** to provide the processed task to another processing cluster **214** for further processing or to store the processed task in an L2 cache, local parallel processor memory, or system memory via the memory crossbar **216**. A preROP **242** (pre-raster operations unit) is configured to receive data from graphics multiprocessor **234**, direct data to ROP units, which may be located with partition units as described herein (e.g., partition units **220**A-**220**N of FIG. **2**). The preROP **242** unit can perform optimizations for color blending, organize pixel color data, and perform address translations.

It will be appreciated that the core architecture described herein is illustrative and that variations and modifications are possible. Any number of processing units, e.g., graphics multiprocessor **234**, texture units **236**, preROPs **242**, etc., may be included within a processing cluster **214**. Further, while only one processing cluster **214** is shown, a parallel processing unit as described herein may include any number of instances of the processing cluster **214**. In one embodiment, each processing cluster **214** can be configured to operate independently of other processing clusters **214** using separate and distinct processing units, L1 caches, etc.

FIG. **2**D shows a graphics multiprocessor **234**, according to one embodiment. In such embodiment the graphics multiprocessor **234** couples with the pipeline manager **232** of the processing cluster **214**. The graphics multiprocessor **234** has an execution pipeline including but not limited to an instruction cache **252**, an instruction unit **254**, an address mapping unit **256**, a register file **258**, one or more general purpose graphics processing unit (GPGPU) cores **262**, and one or more load/store units **266**. The GPGPU cores **262** and load/store units **266** are coupled with cache memory **272** and shared memory **270** via a memory and cache interconnect **268**.

In one embodiment, the instruction cache **252** receives a stream of instructions to execute from the pipeline manager **232**. The instructions are cached in the instruction cache **252** and dispatched for execution by the instruction unit **254**. The instruction unit **254** can dispatch instructions as thread groups (e.g., warps), with each thread of the thread group assigned to a different execution unit within GPGPU core **262**. An instruction can access any of a local, shared, or global address space by specifying an address within a unified address space. The address mapping unit **256** can be used to translate addresses in the unified address space into a distinct memory address that can be accessed by the load/store units **266**.

The register file **258** provides a set of registers for the functional units of the graphics multiprocessor **324**. The register file **258** provides temporary storage for operands connected to the data paths of the functional units (e.g., GPGPU cores **262**, load/store units **266**) of the graphics multiprocessor **324**. In one embodiment, the register file **258** is divided between each of the functional units such that each functional unit is allocated a dedicated portion of the register file **258**. In one embodiment, the register file **258** is divided between the different warps being executed by the graphics multiprocessor **324**.

The GPGPU cores **262** can each include floating point units (FPUs) and/or integer arithmetic logic units (ALUs) that are used to execute instructions of the graphics multiprocessor **324**. The GPGPU cores **262** can be similar in architecture or can differ in architecture, according to embodiments. For example and in one embodiment, a first portion of the GPGPU cores **262** include a single precision FPU and an integer ALU while a second portion of the GPGPU cores include a double precision FPU. In one embodiment the FPUs can implement the IEEE 754-2008 standard for floating point arithmetic or enable variable precision floating point arithmetic. The graphics multipro-

cessor **324** can additionally include one or more fixed function or special function units to perform specific functions such as copy rectangle or pixel blending operations. In one embodiment one or more of the GPGPU cores can also include fixed or special function logic.

In one embodiment the GPGPU cores **262** include SIMD logic capable of performing a single instruction on multiple sets of data. In one embodiment GPGPU cores **262** can physically execute SIMD4, SIMD8, and SIMD16 instructions and logically execute SIMD1, SIMD2, and SIMD32 instructions. The SIMD instructions for the GPGPU cores can be generated at compile time by a shader compiler or automatically generated when executing programs written and compiled for single program multiple data (SPMD) or SIMT architectures. Multiple threads of a program configured for the SIMT execution model can executed via a single SIMD instruction. For example and in one embodiment, eight SIMT threads that perform the same or similar operations can be executed in parallel via a single SIMD8 logic unit.

The memory and cache interconnect **268** is an interconnect network that connects each of the functional units of the graphics multiprocessor **324** to the register file **258** and to the shared memory **270**. In one embodiment, the memory and cache interconnect **268** is a crossbar interconnect that allows the load/store unit **266** to implement load and store operations between the shared memory **270** and the register file **258**. The register file **258** can operate at the same frequency as the GPGPU cores **262**, thus data transfer between the GPGPU cores **262** and the register file **258** is very low latency. The shared memory **270** can be used to enable communication between threads that execute on the functional units within the graphics multiprocessor **234**. The cache memory **272** can be used as a data cache for example, to cache texture data communicated between the functional units and the texture unit **236**. The shared memory **270** can also be used as a program managed cached. Threads executing on the GPGPU cores **262** can programmatically store data within the shared memory in addition to the automatically cached data that is stored within the cache memory **272**.

FIGS. **3**A-**3**B illustrate additional graphics multiprocessors, according to embodiments. The illustrated graphics multiprocessors **325**, **350** are variants of the graphics multiprocessor **234** of FIG. **2**C. The illustrated graphics multiprocessors **325**, **350** can be configured as a streaming multiprocessor (SM) capable of simultaneous execution of a large number of execution threads.

FIG. **3**A shows a graphics multiprocessor **325** according to an additional embodiment. The graphics multiprocessor **325** includes multiple additional instances of execution resource units relative to the graphics multiprocessor **234** of FIG. **2**D. For example, the graphics multiprocessor **325** can include multiple instances of the instruction unit **332**A-**332**B, register file **334**A-**334**B, and texture unit(s) **344**A-**344**B. The graphics multiprocessor **325** also includes multiple sets of graphics or compute execution units (e.g., GPGPU core **336**A-**336**B, GPGPU core **337**A-**337**B, GPGPU core **338**A-**338**B) and multiple sets of load/store units **340**A-**340**B. In one embodiment the execution resource units have a common instruction cache **330**, texture and/or data cache memory **342**, and shared memory **346**.

The various components can communicate via an interconnect fabric **327**. In one embodiment the interconnect fabric **327** includes one or more crossbar switches to enable communication between the various components of the graphics multiprocessor **325**. In one embodiment the interconnect fabric **327** is a separate, high-speed network fabric layer upon which each component of the graphics multiprocessor **325** is stacked. The components of the graphics multiprocessor **325** communicate with remote components via the interconnect fabric **327**. For example, the GPGPU cores **336**A-**336**B, **337**A-**337**B, and **3378**A-**338**B can each communicate with shared memory **346** via the interconnect fabric **327**. The interconnect fabric **327** can arbitrate communication within the graphics multiprocessor **325** to ensure a fair bandwidth allocation between components.

FIG. **3**B shows a graphics multiprocessor **350** according to an additional embodiment. The graphics processor includes multiple sets of execution resources **356**A-**356**D, where each set of execution resource includes multiple instruction units, register files, GPGPU cores, and load store units, as illustrated in FIG. **2**D and FIG. **3**A. The execution resources **356**A-**356**D can work in concert with texture unit(s) **360**A-**360**D for texture operations, while sharing an instruction cache **354**, and shared memory **362**. In one embodiment the execution resources **356**A-**356**D can share an instruction cache **354** and shared memory **362**, as well as multiple instances of a texture and/or data cache memory **358**A-**358**B. The various components can communicate via an interconnect fabric **352** similar to the interconnect fabric **327** of FIG. **3**A.

Persons skilled in the art will understand that the architecture described in FIGS. **1**, **2**A-**2**D, and **3**A-**3**B are descriptive and not limiting as to the scope of the present embodiments. Thus, the techniques described herein may be implemented on any properly configured processing unit, including, without limitation, one or more mobile application processors, one or more desktop or server central processing units (CPUs) including multi-core CPUs, one or more parallel processing units, such as the parallel processing unit **202** of FIG. **2**, as well as one or more graphics processors or special purpose processing units, without departure from the scope of the embodiments described herein.

In some embodiments a parallel processor or GPGPU as described herein is communicatively coupled to host/processor cores to accelerate graphics operations, machine-learning operations, pattern analysis operations, and various general purpose GPU (GPGPU) functions. The GPU may be communicatively coupled to the host processor/cores over a bus or other interconnect (e.g., a high speed interconnect such as PCIe or NVLink). In other embodiments, the GPU may be integrated on the same package or chip as the cores and communicatively coupled to the cores over an internal processor bus/interconnect (i.e., internal to the package or chip). Regardless of the manner in which the GPU is connected, the processor cores may allocate work to the GPU in the form of sequences of commands/instructions contained in a work descriptor. The GPU then uses dedicated circuitry/logic for efficiently processing these commands/instructions.

Techniques for GPU to Host Processor Interconnection

FIG. **4**A illustrates an exemplary architecture in which a plurality of GPUs **410**-**413** are communicatively coupled to a plurality of multi-core processors **405**-**406** over high-speed links **440**-**443** (e.g., buses, point-to-point interconnects, etc.). In one embodiment, the high-speed links **440**-**443** support a communication throughput of 4 GB/s, 30 GB/s, 80 GB/s or higher, depending on the implementation. Various interconnect protocols may be used including, but not limited to, PCIe 4.0 or 5.0 and NVLink 2.0. However, the underlying principles of the invention are not limited to any particular communication protocol or throughput.

In addition, in one embodiment, two or more of the GPUs **410-413** are interconnected over high-speed links **444-445**, which may be implemented using the same or different protocols/links than those used for high-speed links **440-443**. Similarly, two or more of the multi-core processors **405-406** may be connected over high speed link **433** which may be symmetric multi-processor (SMP) buses operating at 20 GB/s, 30 GB/s, 120 GB/s or higher. Alternatively, all communication between the various system components shown in FIG. **4**A may be accomplished using the same protocols/links (e.g., over a common interconnection fabric). As mentioned, however, the underlying principles of the invention are not limited to any particular type of interconnect technology.

In one embodiment, each multi-core processor **405-406** is communicatively coupled to a processor memory **401-402**, via memory interconnects **430-431**, respectively, and each GPU **410-413** is communicatively coupled to GPU memory **420-423** over GPU memory interconnects **450-453**, respectively. The memory interconnects **430-431** and **450-453** may utilize the same or different memory access technologies. By way of example, and not limitation, the processor memories **401-402** and GPU memories **420-423** may be volatile memories such as dynamic random access memories (DRAMs) (including stacked DRAMs), Graphics DDR SDRAM (GDDR) (e.g., GDDR5, GDDR6), or High Bandwidth Memory (HBM) and/or may be non-volatile memories such as 3D XPoint or Nano-Ram. In one embodiment, some portion of the memories may be volatile memory and another portion may be non-volatile memory (e.g., using a two-level memory (2LM) hierarchy).

As described below, although the various processors **405-406** and GPUs **410-413** may be physically coupled to a particular memory **401-402**, **420-423**, respectively, a unified memory architecture may be implemented in which the same virtual system address space (also referred to as the "effective address" space) is distributed among all of the various physical memories. For example, processor memories **401-402** may each comprise 64 GB of the system memory address space and GPU memories **420-423** may each comprise 32 GB of the system memory address space (resulting in a total of 256 GB addressable memory in this example).

FIG. **4**B illustrates additional details for an interconnection between a multi-core processor **407** and a graphics acceleration module **446** in accordance with one embodiment. The graphics acceleration module **446** may include one or more GPU chips integrated on a line card which is coupled to the processor **407** via the high-speed link **440**. Alternatively, the graphics acceleration module **446** may be integrated on the same package or chip as the processor **407**.

The illustrated processor **407** includes a plurality of cores **460**A-**460**D, each with a translation lookaside buffer **461**A-**461**D and one or more caches **462**A-**462**D. The cores may include various other components for executing instructions and processing data which are not illustrated to avoid obscuring the underlying principles of the invention (e.g., instruction fetch units, branch prediction units, decoders, execution units, reorder buffers, etc.). The caches **462**A-**462**D may comprise level 1 (L1) and level 2 (L2) caches. In addition, one or more shared caches **426** may be included in the caching hierarchy and shared by sets of the cores **460**A-**460**D. For example, one embodiment of the processor **407** includes 24 cores, each with its own L1 cache, twelve shared L2 caches, and twelve shared L3 caches. In this embodiment, one of the L2 and L3 caches are shared by two adjacent cores. The processor **407** and the graphics accelerator integration module **446** connect with system memory **441**, which may include processor memories **401-402**

Coherency is maintained for data and instructions stored in the various caches **462**A-**462**D, **456** and system memory **441** via inter-core communication over a coherence bus **464**. For example, each cache may have cache coherency logic/circuitry associated therewith to communicate to over the coherence bus **464** in response to detected reads or writes to particular cache lines. In one implementation, a cache snooping protocol is implemented over the coherence bus **464** to snoop cache accesses. Cache snooping/coherency techniques are well understood by those of skill in the art and will not be described in detail here to avoid obscuring the underlying principles of the invention.

In one embodiment, a proxy circuit **425** communicatively couples the graphics acceleration module **446** to the coherence bus **464**, allowing the graphics acceleration module **446** to participate in the cache coherence protocol as a peer of the cores. In particular, an interface **435** provides connectivity to the proxy circuit **425** over high-speed link **440** (e.g., a PCIe bus, NVLink, etc.) and an interface **437** connects the graphics acceleration module **446** to the link **440**.

In one implementation, an accelerator integration circuit **436** provides cache management, memory access, context management, and interrupt management services on behalf of a plurality of graphics processing engines **431**, **432**, N of the graphics acceleration module **446**. The graphics processing engines **431**, **432**, N may each comprise a separate graphics processing unit (GPU). Alternatively, the graphics processing engines **431**, **432**, N may comprise different types of graphics processing engines within a GPU such as graphics execution units, media processing engines (e.g., video encoders/decoders), samplers, and blit engines. In other words, the graphics acceleration module may be a GPU with a plurality of graphics processing engines **431-432**, N or the graphics processing engines **431-432**, N may be individual GPUs integrated on a common package, line card, or chip.

In one embodiment, the accelerator integration circuit **436** includes a memory management unit (MMU) **439** for performing various memory management functions such as virtual-to-physical memory translations (also referred to as effective-to-real memory translations) and memory access protocols for accessing system memory **441**. The MMU **439** may also include a translation lookaside buffer (TLB) (not shown) for caching the virtual/effective to physical/real address translations. In one implementation, a cache **438** stores commands and data for efficient access by the graphics processing engines **431-432**, N. In one embodiment, the data stored in cache **438** and graphics memories **433-434**, N is kept coherent with the core caches **462**A-**462**D, **456** and system memory **411**. As mentioned, this may be accomplished via proxy circuit **425** which takes part in the cache coherency mechanism on behalf of cache **438** and memories **433-434**, N (e.g., sending updates to the cache **438** related to modifications/accesses of cache lines on processor caches **462**A-**462**D, **456** and receiving updates from the cache **438**).

A set of registers **445** store context data for threads executed by the graphics processing engines **431-432**, N and a context management circuit **448** manages the thread contexts. For example, the context management circuit **448** may perform save and restore operations to save and restore contexts of the various threads during contexts switches (e.g., where a first thread is saved and a second thread is stored so that the second thread can be execute by a graphics processing engine). For example, on a context switch, the

context management circuit **448** may store current register values to a designated region in memory (e.g., identified by a context pointer). It may then restore the register values when returning to the context. In one embodiment, an interrupt management circuit **447** receives and processes interrupts received from system devices.

In one implementation, virtual/effective addresses from a graphics processing engine **431** are translated to real/physical addresses in system memory **411** by the MMU **439**. One embodiment of the accelerator integration circuit **436** supports multiple (e.g., 4, 8, 16) graphics accelerator modules **446** and/or other accelerator devices. The graphics accelerator module **446** may be dedicated to a single application executed on the processor **407** or may be shared between multiple applications. In one embodiment, a virtualized graphics execution environment is presented in which the resources of the graphics processing engines **431-432**, N are shared with multiple applications or virtual machines (VMs). The resources may be subdivided into "slices" which are allocated to different VMs and/or applications based on the processing requirements and priorities associated with the VMs and/or applications.

Thus, the accelerator integration circuit acts as a bridge to the system for the graphics acceleration module **446** and provides address translation and system memory cache services. In addition, the accelerator integration circuit **436** may provide virtualization facilities for the host processor to manage virtualization of the graphics processing engines, interrupts, and memory management.

Because hardware resources of the graphics processing engines **431-432**, N are mapped explicitly to the real address space seen by the host processor **407**, any host processor can address these resources directly using an effective address value. One function of the accelerator integration circuit **436**, in one embodiment, is the physical separation of the graphics processing engines **431-432**, N so that they appear to the system as independent units.

As mentioned, in the illustrated embodiment, one or more graphics memories **433-434**, M are coupled to each of the graphics processing engines **431-432**, N, respectively. The graphics memories **433-434**, M store instructions and data being processed by each of the graphics processing engines **431-432**, N. The graphics memories **433-434**, M may be volatile memories such as DRAMs (including stacked DRAMs), GDDR memory (e.g., GDDR5, GDDR6), or HBM, and/or may be non-volatile memories such as 3D XPoint or Nano-Ram.

In one embodiment, to reduce data traffic over link **440**, biasing techniques are used to ensure that the data stored in graphics memories **433-434**, M is data which will be used most frequently by the graphics processing engines **431-432**, N and preferably not used by the cores **460**A-**460**D (at least not frequently). Similarly, the biasing mechanism attempts to keep data needed by the cores (and preferably not the graphics processing engines **431-432**, N) within the caches **462**A-**462**D, **456** of the cores and system memory **411**.

FIG. **4**C illustrates another embodiment in which the accelerator integration circuit **436** is integrated within the processor **407**. In this embodiment, the graphics processing engines **431-432**, N communicate directly over the high-speed link **440** to the accelerator integration circuit **436** via interface **437** and interface **435** (which, again, may be utilize any form of bus or interface protocol). The accelerator integration circuit **436** may perform the same operations as those described with respect to FIG. **4**B, but potentially at a higher throughput given its close proximity to the coherency bus **462** and caches **462**A-**462**D, **426**.

One embodiment supports different programming models including a dedicated-process programming model (no graphics acceleration module virtualization) and shared programming models (with virtualization). The latter may include programming models which are controlled by the accelerator integration circuit **436** and programming models which are controlled by the graphics acceleration module **446**.

In one embodiment of the dedicated process model, graphics processing engines **431-432**, N are dedicated to a single application or process under a single operating system. The single application can funnel other application requests to the graphics engines **431-432**, N, providing virtualization within a VM/partition.

In the dedicated-process programming models, the graphics processing engines **431-432**, N, may be shared by multiple VM/application partitions. The shared models require a system hypervisor to virtualize the graphics processing engines **431-432**, N to allow access by each operating system. For single-partition systems without a hypervisor, the graphics processing engines **431-432**, N are owned by the operating system. In both cases, the operating system can virtualize the graphics processing engines **431-432**, N to provide access to each process or application.

For the shared programming model, the graphics acceleration module **446** or an individual graphics processing engine **431-432**, N selects a process element using a process handle. In one embodiment, process elements are stored in system memory **411** and are addressable using the effective address to real address translation techniques described herein. The process handle may be an implementation-specific value provided to the host process when registering its context with the graphics processing engine **431-432**, N (that is, calling system software to add the process element to the process element linked list). The lower 16-bits of the process handle may be the offset of the process element within the process element linked list.

FIG. **4**D illustrates an exemplary accelerator integration slice **490**. As used herein, a "slice" comprises a specified portion of the processing resources of the accelerator integration circuit **436**. Application effective address space **482** within system memory **411** stores process elements **483**. In one embodiment, the process elements **483** are stored in response to GPU invocations **481** from applications **480** executed on the processor **407**. A process element **483** contains the process state for the corresponding application **480**. A work descriptor (WD) **484** contained in the process element **483** can be a single job requested by an application or may contain a pointer to a queue of jobs. In the latter case, the WD **484** is a pointer to the job request queue in the application's address space **482**.

The graphics acceleration module **446** and/or the individual graphics processing engines **431-432**, N can be shared by all or a subset of the processes in the system. Embodiments of the invention include an infrastructure for setting up the process state and sending a WD **484** to a graphics acceleration module **446** to start a job in a virtualized environment.

In one implementation, the dedicated-process programming model is implementation-specific. In this model, a single process owns the graphics acceleration module **446** or an individual graphics processing engine **431**. Because the graphics acceleration module **446** is owned by a single process, the hypervisor initializes the accelerator integration circuit **436** for the owning partition and the operating system

initializes the accelerator integration circuit **436** for the owning process at the time when the graphics acceleration module **446** is assigned.

In operation, a WD fetch unit **491** in the accelerator integration slice **490** fetches the next WD **484** which includes an indication of the work to be done by one of the graphics processing engines of the graphics acceleration module **446**. Data from the WD **484** may be stored in registers **445** and used by the MMU **439**, interrupt management circuit **447** and/or context management circuit **446** as illustrated. For example, one embodiment of the MMU **439** includes segment/page walk circuitry for accessing segment/page tables **486** within the OS virtual address space **485**. The interrupt management circuit **447** may process interrupt events **492** received from the graphics acceleration module **446**. When performing graphics operations, an effective address **493** generated by a graphics processing engine **431-432**, N is translated to a real address by the MMU **439**.

In one embodiment, the same set of registers **445** are duplicated for each graphics processing engine **431-432**, N and/or graphics acceleration module **446** and may be initialized by the hypervisor or operating system. Each of these duplicated registers may be included in an accelerator integration slice **490**. Exemplary registers that may be initialized by the hypervisor are shown in Table 1.

TABLE 1

Hypervisor Initialized Registers	
1	Slice Control Register
2	Real Address (RA) Scheduled Processes Area Pointer
3	Authority Mask Override Register
4	Interrupt Vector Table Entry Offset
5	Interrupt Vector Table Entry Limit
6	State Register
7	Logical Partition ID
8	Real address (RA) Hypervisor Accelerator Utilization Record Pointer
9	Storage Description Register

Exemplary registers that may be initialized by the operating system are shown in Table 2.

TABLE 2

Operating System Initialized Registers	
1	Process and Thread Identification
2	Effective Address (EA) Context Save/Restore Pointer
3	Virtual Address (VA) Accelerator Utilization Record Pointer
4	Virtual Address (VA) Storage Segment Table Pointer
5	Authority Mask
6	Work descriptor

In one embodiment, each WD **484** is specific to a particular graphics acceleration module **446** and/or graphics processing engine **431-432**, N. It contains all the information a graphics processing engine **431-432**, N requires to do its work or it can be a pointer to a memory location where the application has set up a command queue of work to be completed.

FIG. **4**E illustrates additional details for one embodiment of a shared model. This embodiment includes a hypervisor real address space **498** in which a process element list **499** is stored. The hypervisor real address space **498** is accessible via a hypervisor **496** which virtualizes the graphics acceleration module engines for the operating system **495**.

The shared programming models allow for all or a subset of processes from all or a subset of partitions in the system to use a graphics acceleration module **446**. There are two programming models where the graphics acceleration module **446** is shared by multiple processes and partitions: time-sliced shared and graphics directed shared.

In this model, the system hypervisor **496** owns the graphics acceleration module **446** and makes its function available to all operating systems **495**. For a graphics acceleration module **446** to support virtualization by the system hypervisor **496**, the graphics acceleration module **446** may adhere to the following requirements: 1) An application's job request must be autonomous (that is, the state does not need to be maintained between jobs), or the graphics acceleration module **446** must provide a context save and restore mechanism. 2) An application's job request is guaranteed by the graphics acceleration module **446** to complete in a specified amount of time, including any translation faults, or the graphics acceleration module **446** provides the ability to preempt the processing of the job. 3) The graphics acceleration module **446** must be guaranteed fairness between processes when operating in the directed shared programming model.

In one embodiment, for the shared model, the application **480** is required to make an operating system **495** system call with a graphics acceleration module **446** type, a work descriptor (WD), an authority mask register (AMR) value, and a context save/restore area pointer (CSRP). The graphics acceleration module **446** type describes the targeted acceleration function for the system call. The graphics acceleration module **446** type may be a system-specific value. The WD is formatted specifically for the graphics acceleration module **446** and can be in the form of a graphics acceleration module **446** command, an effective address pointer to a user-defined structure, an effective address pointer to a queue of commands, or any other data structure to describe the work to be done by the graphics acceleration module **446**. In one embodiment, the AMR value is the AMR state to use for the current process. The value passed to the operating system is similar to an application setting the AMR. If the accelerator integration circuit **436** and graphics acceleration module **446** implementations do not support a User Authority Mask Override Register (UAMOR), the operating system may apply the current UAMOR value to the AMR value before passing the AMR in the hypervisor call. The hypervisor **496** may optionally apply the current Authority Mask Override Register (AMOR) value before placing the AMR into the process element **483**. In one embodiment, the CSRP is one of the registers **445** containing the effective address of an area in the application's address space **482** for the graphics acceleration module **446** to save and restore the context state. This pointer is optional if no state is required to be saved between jobs or when a job is preempted. The context save/restore area may be pinned system memory.

Upon receiving the system call, the operating system **495** may verify that the application **480** has registered and been given the authority to use the graphics acceleration module **446**. The operating system **495** then calls the hypervisor **496** with the information shown in Table 3.

TABLE 3

OS to Hypervisor Call Parameters	
1	A work descriptor (WD)
2	An Authority Mask Register (AMR) value (potentially masked).
3	An effective address (EA) Context Save/Restore Area Pointer (CSRP)
4	A process ID (PID) and optional thread ID (TID)
5	A virtual address (VA) accelerator utilization record pointer (AURP)
6	The virtual address of the storage segment table pointer (SSTP)
7	A logical interrupt service number (LISN)

Upon receiving the hypervisor call, the hypervisor **496** verifies that the operating system **495** has registered and been given the authority to use the graphics acceleration module **446**. The hypervisor **496** then puts the process element **483** into the process element linked list for the corresponding graphics acceleration module **446** type. The process element may include the information shown in Table 4.

TABLE 4

Process Element Information	
1	A work descriptor (WD)
2	An Authority Mask Register (AMR) value (potentially masked).
3	An effective address (EA) Context Save/Restore Area Pointer (CSRP)
4	A process ID (PID) and optional thread ID (TID)
5	A virtual address (VA) accelerator utilization record pointer (AURP)
6	The virtual address of the storage segment table pointer (SSTP)
7	A logical interrupt service number (LISN)
8	Interrupt vector table, derived from the hypervisor call parameters.
9	A state register (SR) value
10	A logical partition ID (LPID)
11	A real address (RA) hypervisor accelerator utilization record pointer
12	The Storage Descriptor Register (SDR)

In one embodiment, the hypervisor initializes a plurality of accelerator integration slice **490** registers **445**.

As illustrated in FIG. **4**F, one embodiment of the invention employs a unified memory addressable via a common virtual memory address space used to access the physical processor memories **401-402** and GPU memories **420-423**. In this implementation, operations executed on the GPUs **410-413** utilize the same virtual/effective memory address space to access the processors memories **401-402** and vice versa, thereby simplifying programmability. In one embodiment, a first portion of the virtual/effective address space is allocated to the processor memory **401**, a second portion to the second processor memory **402**, a third portion to the GPU memory **420**, and so on. The entire virtual/effective memory space (sometimes referred to as the effective address space) is thereby distributed across each of the processor memories **401-402** and GPU memories **420-423**, allowing any processor or GPU to access any physical memory with a virtual address mapped to that memory.

In one embodiment, bias/coherence management circuitry **494**A-**494**E within one or more of the MMUs **439**A-**439**E ensures cache coherence between the caches of the host processors (e.g., **405**) and the GPUs **410-413** and implements biasing techniques indicating the physical memories in which certain types of data should be stored. While multiple instances of bias/coherence management circuitry **494**A-**494**E are illustrated in FIG. **4**F, the bias/coherence circuitry may be implemented within the MMU of one or more host processors **405** and/or within the accelerator integration circuit **436**.

One embodiment allows GPU-attached memory **420-423** to be mapped as part of system memory, and accessed using shared virtual memory (SVM) technology, but without suffering the typical performance drawbacks associated with full system cache coherence. The ability to GPU-attached memory **420-423** to be accessed as system memory without onerous cache coherence overhead provides a beneficial operating environment for GPU offload. This arrangement allows the host processor **405** software to setup operands and access computation results, without the overhead of tradition I/O DMA data copies. Such traditional copies involve driver calls, interrupts and memory mapped I/O (MMIO) accesses that are all inefficient relative to simple memory accesses. At the same time, the ability to access GPU attached memory **420-423** without cache coherence overheads can be critical to the execution time of an offloaded computation. In cases with substantial streaming write memory traffic, for example, cache coherence overhead can significantly reduce the effective write bandwidth seen by a GPU **410-413**. The efficiency of operand setup, the efficiency of results access, and the efficiency of GPU computation all play a role in determining the effectiveness of GPU offload.

In one implementation, the selection of between GPU bias and host processor bias is driven by a bias tracker data structure. A bias table may be used, for example, which may be a page-granular structure (i.e., controlled at the granularity of a memory page) that includes 1 or 2 bits per GPU-attached memory page. The bias table may be implemented in a stolen memory range of one or more GPU-attached memories **420-423**, with or without a bias cache in the GPU **410-413** (e.g., to cache frequently/recently used entries of the bias table). Alternatively, the entire bias table may be maintained within the GPU.

In one implementation, the bias table entry associated with each access to the GPU-attached memory **420-423** is accessed prior the actual access to the GPU memory, causing the following operations. First, local requests from the GPU **410-413** that find their page in GPU bias are forwarded directly to a corresponding GPU memory **420-423**. Local requests from the GPU that find their page in host bias are forwarded to the processor **405** (e.g., over a high-speed link as discussed above). In one embodiment, requests from the processor **405** that find the requested page in host processor bias complete the request like a normal memory read. Alternatively, requests directed to a GPU-biased page may be forwarded to the GPU **410-413**. The GPU may then transition the page to a host processor bias if it is not currently using the page.

The bias state of a page can be changed either by a software-based mechanism, a hardware-assisted software-based mechanism, or, for a limited set of cases, a purely hardware-based mechanism.

One mechanism for changing the bias state employs an API call (e.g. OpenCL), which, in turn, calls the GPU's device driver which, in turn, sends a message (or enqueues a command descriptor) to the GPU directing it to change the bias state and, for some transitions, perform a cache flushing operation in the host. The cache flushing operation is

required for a transition from host processor **405** bias to GPU bias, but is not required for the opposite transition.

In one embodiment, cache coherency is maintained by temporarily rendering GPU-biased pages uncacheable by the host processor **405**. To access these pages, the processor **405** may request access from the GPU **410** which may or may not grant access right away, depending on the implementation. Thus, to reduce communication between the processor **405** and GPU **410** it is beneficial to ensure that GPU-biased pages are those which are required by the GPU but not the host processor **405** and vice versa.

Graphics Processing Pipeline

FIG. **5** illustrates a graphics processing pipeline **500**, according to an embodiment. In one embodiment a graphics processor can implement the illustrated graphics processing pipeline **500**. The graphics processor can be included within the parallel processing subsystems as described herein, such as the parallel processor **200** of FIG. **2**, which, in one embodiment, is a variant of the parallel processor(s) **112** of FIG. **1**. The various parallel processing systems can implement the graphics processing pipeline **500** via one or more instances of the parallel processing unit (e.g., parallel processing unit **202** of FIG. **2**) as described herein. For example, a shader unit (e.g., graphics multiprocessor **234** of FIG. **3**) may be configured to perform the functions of one or more of a vertex processing unit **504**, a tessellation control processing unit **508**, a tessellation evaluation processing unit **512**, a geometry processing unit **516**, and a fragment/pixel processing unit **524**. The functions of data assembler **502**, primitive assemblers **506**, **514**, **518**, tessellation unit **510**, rasterizer **522**, and raster operations unit **526** may also be performed by other processing engines within a processing cluster (e.g., processing cluster **214** of FIG. **3**) and a corresponding partition unit (e.g., partition unit **220**A-**220**N of FIG. **2**). The graphics processing pipeline **500** may also be implemented using dedicated processing units for one or more functions. In one embodiment, one or more portions of the graphics processing pipeline **500** can be performed by parallel processing logic within a general purpose processor (e.g., CPU). In one embodiment, one or more portions of the graphics processing pipeline **500** can access on-chip memory (e.g., parallel processor memory **222** as in FIG. **2**) via a memory interface **528**, which may be an instance of the memory interface **218** of FIG. **2**.

In one embodiment the data assembler **502** is a processing unit that collects vertex data for surfaces and primitives. The data assembler **502** then outputs the vertex data, including the vertex attributes, to the vertex processing unit **504**. The vertex processing unit **504** is a programmable execution unit that executes vertex shader programs, lighting and transforming vertex data as specified by the vertex shader programs. The vertex processing unit **504** reads data that is stored in cache, local or system memory for use in processing the vertex data and may be programmed to transform the vertex data from an object-based coordinate representation to a world space coordinate space or a normalized device coordinate space.

A first instance of a primitive assembler **506** receives vertex attributes from the vertex processing unit **50**. The primitive assembler **506** readings stored vertex attributes as needed and constructs graphics primitives for processing by tessellation control processing unit **508**. The graphics primitives include triangles, line segments, points, patches, and so forth, as supported by various graphics processing application programming interfaces (APIs).

The tessellation control processing unit **508** treats the input vertices as control points for a geometric patch. The control points are transformed from an input representation from the patch (e.g., the patch's bases) to a representation that is suitable for use in surface evaluation by the tessellation evaluation processing unit **512**. The tessellation control processing unit **508** can also compute tessellation factors for edges of geometric patches. A tessellation factor applies to a single edge and quantifies a view-dependent level of detail associated with the edge. A tessellation unit **510** is configured to receive the tessellation factors for edges of a patch and to tessellate the patch into multiple geometric primitives such as line, triangle, or quadrilateral primitives, which are transmitted to a tessellation evaluation processing unit **512**. The tessellation evaluation processing unit **512** operates on parameterized coordinates of the subdivided patch to generate a surface representation and vertex attributes for each vertex associated with the geometric primitives.

A second instance of a primitive assembler **514** receives vertex attributes from the tessellation evaluation processing unit **512**, reading stored vertex attributes as needed, and constructs graphics primitives for processing by the geometry processing unit **516**. The geometry processing unit **516** is a programmable execution unit that executes geometry shader programs to transform graphics primitives received from primitive assembler **514** as specified by the geometry shader programs. In one embodiment the geometry processing unit **516** is programmed to subdivide the graphics primitives into one or more new graphics primitives and calculate parameters used to rasterize the new graphics primitives.

In some embodiments the geometry processing unit **516** can add or delete elements in the geometry stream. The geometry processing unit **516** outputs the parameters and vertices specifying new graphics primitives to primitive assembler **518**. The primitive assembler **518** receives the parameters and vertices from the geometry processing unit **516** and constructs graphics primitives for processing by a viewport scale, cull, and clip unit **520**. The geometry processing unit **516** reads data that is stored in parallel processor memory or system memory for use in processing the geometry data. The viewport scale, cull, and clip unit **520** performs clipping, culling, and viewport scaling and outputs processed graphics primitives to a rasterizer **522**.

The rasterizer **522** can perform depth culling and other depth-based optimizations. The rasterizer **522** also performs scan conversion on the new graphics primitives to generate fragments and output those fragments and associated coverage data to the fragment/pixel processing unit **524**. The fragment/pixel processing unit **524** is a programmable execution unit that is configured to execute fragment shader programs or pixel shader programs. The fragment/pixel processing unit **524** transforming fragments or pixels received from rasterizer **522**, as specified by the fragment or pixel shader programs. For example, the fragment/pixel processing unit **524** may be programmed to perform operations included but not limited to texture mapping, shading, blending, texture correction and perspective correction to produce shaded fragments or pixels that are output to a raster operations unit **526**. The fragment/pixel processing unit **524** can read data that is stored in either the parallel processor memory or the system memory for use when processing the fragment data. Fragment or pixel shader programs may be configured to shade at sample, pixel, tile, or other granularities depending on the sampling rate configured for the processing units.

The raster operations unit **526** is a processing unit that performs raster operations including, but not limited to

stencil, z test, blending, and the like, and outputs pixel data as processed graphics data to be stored in graphics memory (e.g., parallel processor memory **222** as in FIG. **2**, and/or system memory **104** as in FIG. **1**, to be displayed on the one or more display device(s) **110** or for further processing by one of the one or more processor(s) **102** or parallel processor(s) **112**. In some embodiments the raster operations unit **526** is configured to compress z or color data that is written to memory and decompress z or color data that is read from memory.

System, Apparatus and Method for Segmenting a Memory Array

In various embodiments, a memory array can be provided with a hybrid design including both domino-based logic and static-based logic to reduce read power consumption. Note in a domino-based logic, a logical function is implemented using either n-channel metal oxide semiconductor devices or p-channel metal oxide semiconductor devices. In static-based logic, a logic function is implemented using fully complementary n-channel and p-channel metal oxide semiconductor devices. In addition, the memory array may be implemented with a segmented design where certain portions of the memory array have shorter interconnect distances to the static-based logic portions such that reduced power consumption can be realized during read operations. To further leverage this property, embodiments provide mechanisms to enable storage of more frequently accessed information in these so-called low order memory portions.

FIG. **6** is a schematic diagram of a memory circuit **600** in accordance with an embodiment. Memory circuit **600** may include a plurality of memory bitcells **602**, a representative bitcell $\mathbf{602}_0$ shown in further detail. Individual bitcells **602** may store one or more bits of data. Bitcells **602** may be coupled to a supply node to receive a supply voltage (e.g., Vcc). In some embodiments, bitcells **602** may be register file bitcells of a register file array. For example, bitcells **602** may be 8T bitcells (e.g., including 8 transistors) as shown in FIG. **6**. Or embodiments may include other types of bitcells. As one example, memory **600** may be a single-ported register file having hierarchical local and global read bitlines. In some cases, a single-ended large signal read sensing mechanism and write drivers for dual-N write operation may be provided.

As shown in FIG. **6**, bitcells **602** may include a pair of pull-up transistors (e.g., pull-up transistor **614** and pull-up transistor **616**) and a pair of pull-down transistors (e.g., pull-down transistor **618** and pull-down transistor **620**) arranged as cross-coupled inverters. For example, a drain terminal of pull-up transistor **614** may be coupled to a drain terminal of pull-down transistor **618**, and a drain terminal of pull-up transistor **616** may be coupled to a drain terminal of pull-down transistor **620**. Additionally, gate terminals of pull-up transistor **614** and pull-down transistor **618** may be coupled to a bit node **622** coupled between pull-up transistor **616** and pull-down transistor **620**, and gate terminals of pull-up transistor **616** and pull-down transistor **620** may be coupled to a complementary bit (bitx) node **624** coupled between pull-up transistor **614** and pull-down transistor **618**. In turn, the source terminals of pull-up transistors **614** and **616** may be coupled to a supply node. And, the source terminals of pull-down transistors **618** and **620** may be coupled to a reference voltage node. In some embodiments, pull-up transistors **614** and **616** may be p-type metal oxide semiconductor field effect transistors (PMOS), and pull-down transistors **618** and **620** may be n-type metal oxide semiconductor field effect transistors (NMOS), although other types of transistors may be used in other embodiments.

Individual bitcells **602** may store a value of a bit (e.g., a logic 1 or a logic 0) based on a voltage level at bit node **622** between pull-up transistor **616** and pull-down transistor **620**. For example, bit node **622** may have a first voltage level (e.g., Vcc) to indicate that the bit has a logic 1 value, or a second voltage level (e.g., 0 volts or ground) to indicate that the bit has a logic 0 value.

Additionally, bitx node **624** between pull-up transistor **614** and pull-down transistor **618** may store a complementary bit (bitx) having an opposite logical value from the logical bit value stored at bit node **622**. Accordingly, bitx node **624** may have a second voltage level when bit node **622** has a first voltage level, and may have a first voltage level when bit node **622** has a second voltage level.

A passgate transistor **628** may be coupled between bit node **622** and a write bit line (WBL) **630**. Additionally, a passgate transistor **632** may be coupled between node **624** and a complementary write bit line (WBLX) **634**. The gate terminals of passgate transistors **628** and **632** may be coupled to a write word line (WWL) **636** to receive a WWL signal. The WWL signal may turn on passgate transistors **628** and **632** during a write operation of bitcell **602** and may turn off passgate transistors **628** and **632** during an inactive state of bitcell **602**. During a write operation, bitcell **602** may write a bit to bit node **622** based on a WBL signal received at WBL **630**.

Additionally, bitcell **602** may write a complementary bit to bitx node **624** based on a WBLX signal received at WBLX **634**. In some embodiments, a plurality of bitcells **602** may be associated with a same block (e.g., a column) of a memory device. In some embodiments, a plurality of bitcells **602** may all receive the same WWL signal.

In some embodiments, individual bitcells **602** may further include a read port **638** coupled to bit node **622**. Read port **638** may include NMOS transistors **640** and **642**. A gate terminal of NMOS transistor **640** may be coupled to bit node **622**. NMOS transistor **642** may be coupled between NMOS transistor **640** and a local read bit line (LBL**1**) **644**. A gate terminal of NMOS transistor **642** may be coupled to a read word line (RWL) **646** to receive a RWL signal. The RWL signal may selectively activate a read operation of bitcell **602**. For a write path, a common write driver **680** drives true and inverted versions of write data to all bitcells **602** in a given bundle **605**.

As further illustrated in FIG. **6**, bundle $\mathbf{605}_0$ also includes a pre-charge circuit **650**. In the illustration shown, pre-charge circuit **650** includes a pair of PMOSs, namely PMOS **652** and PMOS **654**. PMOS **652** is controlled by a pre-charge signal, PCH. In turn, PMOS **654** is controlled by an output of an inverter **655**. As further illustrated, the source terminals of PMOSs **652** and **654** couple to the supply voltage node and drain terminals couple to local bitline LBL**1** **644** (which is further coupled to an input of inverter **655**). Pre-charge circuit **650** acts as domino-based logic to pre-charge local bitline LBL**1** **644**. In embodiments herein pre-charge circuit **650** is the only domino logic within bundle $\mathbf{605}_0$.

Still with reference to bundle $\mathbf{605}_0$, the read paths of each of a plurality of bitcells $\mathbf{602}_0$-$\mathbf{602}_{63}$ couple to corresponding local bitlines LBL**1** **644** and LBL**2** **646**. More specifically in the embodiment shown, bitcells $\mathbf{602}_0$-$\mathbf{602}_{31}$ couple to local bitline LBL**1** **644** and bitcells $\mathbf{602}_{32}$-$\mathbf{602}_{63}$ couple to local bitline LBL**2** **646**. While not shown for ease of illustration, understand that another local read bitline LBL **646** may include similar pre-charge circuitry and an inverter.

In the embodiment of FIG. **6**, note that rather than providing combinational logic having dynamic characteris-

tics to combine outputs of the multiple local bitlines, the local bitlines are combined in a local latch circuit **670** of bundle **605** itself, namely a bundle-internal NAND set dominant latch (SDL) **670**. With this arrangement, there is no need for a global pulldown device at an output of a combinational logic (e.g., a NAND gate) to couple local bitlines. Further, there is no single global bitline to which multiple bundles **605** couple, as instead a segmented global bitline arrangement is realized by placing latch circuit **670** within bundle **605**. Instead with the arrangement shown in FIG. **6**, the bundle output (Sdlout_bun0) is a static signal that is provided to a static logic **690**, also referred to as a static multiplexer logic, to select data from one of N bundles. As static logic **690** is implemented with static devices (e.g., selection circuits **692**, **694** and **696**), dynamic power savings are realized. Stated another way, read operations may occur without pre-charging operations of a global bitline using an embodiment.

In contrast, without an embodiment before every read operation local bitlines within bundles and conventional single global bitlines that couple outputs of multiple bundles are pre-charged to a supply voltage level (e.g., Vcc). Hence if the bitlines (reading 1) are evaluated for consecutive read operations, these local and global bitlines keep toggling every phase (one phase pre-charge and next phase discharge), resulting in high dynamic power consumption. In such conventional domino-based memory design, much of the dynamic read power (e.g., 75-80%) results from charging and discharging of global bitlines.

Instead, using an embodiment with a power efficient hybrid memory design, local bitlines LBL**1** **644** and LBL**2** **646** are implemented as domino logic, but combinational logic to merge the outputs of local bitlines each coupled to multiple bitcells within a bundle is implemented as a latch circuit (e.g., in the form of a set dominant latch (SDL) circuit). The output of this SDL circuit is itself a static logic. In the arrangement of FIG. **6**, only the local bitlines (e.g., LBL**1** **644** and LBL**2** **646**) are implemented as domino circuits. In turn, SDL circuit **670** that merges the outputs of LBL**1** **644** and LBL**2** **646** converts the output (Sdlout_bun0) to a static signal, such that there is no need to charge a global bitline every time before a read operation. For a read logic 1 operation, which is the costliest in terms of dynamic power, embodiments may reduce dynamic power of a global bitline by 50%.

Still with reference to FIG. **6**, rather than providing a global bitline to which outputs of multiple bundles couple (e.g., by way of individual pulldown devices within each of the bundles), embodiments provide a segmented global bitline. More specifically as illustrated in FIG. **6**, this segmented global bitline is formed of a plurality of segmented bitlines each associated with a given bundle. In the embodiment of FIG. **6**, the segmented global bitline is formed of a plurality of segmented bitlines $\mathbf{672}_0$-$\mathbf{672}_7$. As seen, each of these segmented bitlines provides an output from a given bundle (and more specifically the merged output of a given SDL circuit **670**) to selection circuitry within static logic **690**.

Still referring to FIG. **6**, static logic **690** includes first level selection circuits **692**, **694** that receive static outputs via the segmented global bitline formed of a plurality of segmented bitlines $\mathbf{672}_0$-$\mathbf{672}_7$ coupled to corresponding bundles **605**. Namely, selection circuit **692** receives static outputs from bundles $\mathbf{605}_0$-$\mathbf{605}_3$ via segmented bitlines $\mathbf{672}_0$-$\mathbf{672}_3$, and selection circuit **694** receives static outputs from bundles $\mathbf{605}_4$-$\mathbf{605}_7$ (via segmented bitlines $\mathbf{672}_4$-$\mathbf{672}_7$). In turn, the outputs of first level selection circuits **692**, **694** are provided to a second level selection circuit **696**, which outputs the read data (Rdout). In an embodiment, first level selection circuit **692** may be controlled by a portion of a read address to output a first selected output from one of bundles $\mathbf{605}_0$-$\mathbf{605}_3$ and first level selection circuit **694** may be controlled by the same portion of the read address to output a second selected output from one of bundles $\mathbf{605}_4$-$\mathbf{605}_7$. Second level selection circuit **696** can be controlled by another portion of the read address (e.g., a most significant bit). Note that all these multiplexer selects may be also ANDed with a read control signal such that the output of these multiplexers only toggles when there is read to the given bundle instance.

With the arrangement of bundles **605** in FIG. **6**, different lengths of traces couple the outputs of different bundles **605** to selection circuitry within static logic **690**. More specifically in the embodiment shown in FIG. **6**, bundles $\mathbf{605}_3$ and $\mathbf{605}_4$ have the shortest distance to couple from their outputs to corresponding selection circuits **692**, **694**. And in turn, bundles $\mathbf{605}_2$ and $\mathbf{605}_5$ have shorter distances to couple to corresponding selection circuits **694**, **694** than bundles $\mathbf{605}_1$ and $\mathbf{605}_6$, which in turn have shorter distances than corresponding bundles $\mathbf{605}_0$ and $\mathbf{605}_7$. As one example, if X is the total distance from bundle $\mathbf{605}_0$ to selection circuit **692**, bundle $\mathbf{605}_1$ will be ¾×, bundle $\mathbf{605}_2$ will be ⅔× and bundle $\mathbf{603}_3$ will be ¼×. In other words, the distance increases ¼× for each successive bundle. As such, bundles having a shortest distance to corresponding selection circuits have a lower power consumption for read operations. Stated another way, with this segmented memory array configuration, read power energy efficiency is greater for the closer-adapted bundles than for the further-adapted bundles.

Understand that while a particular arrangement of bundles with corresponding segmented global bitlines is illustrated in FIG. **6**, many variations and alternatives are possible. For example, different locations of a static logic circuit (and first level selection circuits therein) may occur, such that certain bundles may be equidistant from the static logic circuit. And that it is possible that more than two bundles of a given memory array can be at equal distances with respect to a static logic circuit. And of course other configurations of segmenting a memory array into different portions can occur in other embodiments. In one embodiment, each bundle provides one bit of 64 registers and multiple such bundles are stacked to provide n-bits for a given register. So for example for an array with 64 bits and 512 entries, there may be 8 bundles (8*64=512) of single bits but 64 are stacked to give 64 bits output for every address/register. Also note that while this example is for a bundle of 64 registers, a given bundle can be for 8/16/24/32/40/48/54/64 registers (or even higher).

In a conventional design, whether a read is occurring from bundle $\mathbf{605}_0$ (farthest from a receiver (namely static logic circuit **690**)) or bundle $\mathbf{605}_3$ (closest to the receiver), an entire global bitline capacitance is switching. As such read power is same, regardless of which bundle is being read. Instead embodiments implement a segmented global bitline having segmented bitlines $\mathbf{672}_0$-$\mathbf{672}_7$ by multiplexing 2 or more bundle outputs. With this segmentation, a read operation from bundle $\mathbf{605}_0$ via segmented bitline $\mathbf{672}_0$ may incur a baseline dynamic capacitance of C, a read operation from bundle $\mathbf{605}_1$ via segmented bitline $\mathbf{672}_1$, may incur a Cdyn of (¾)*C and a read operation from bundle $\mathbf{605}_3$ via segmented bitline $\mathbf{672}_3$ may have the Cdyn of (¼)*C, assuming that the bundles are at distances of 100%, 75%, 50%, 25%. More generally, Cdyn is proportional to distance.

Assuming reads are equally divided into different bundles, the Cdyn contribution by providing a segmented

global bitline can be reduced to (½)*C. In this way, the total read power of an energy efficient segmented memory design as described herein can be (½)*(½)*C. As such, embodiments may reduce read power for a logic 1 read operation by approximately 60%. During a logic 0 read operation, a conventional design has one toggle on the global bitline. As such, power savings in an embodiment will be approximately 33%. If equal numbers of read 0 and read 1 operations are assumed, embodiments may result in approximately 50% savings in read power.

There are specific usage scenarios where a certain chunk/block of addresses is read over and over again. With a configuration as in FIG. **6** in which the power consumed to read from certain bundles may be lower, embodiments may selectively store certain information in these low order (shorter distance from the static logic) portions (e.g., bundles) of the memory array. As such, read power may be different for different addresses, such that certain blocks of addresses associated with more frequently accessed information can be stored in lower order portions of the memory (closer to the static logic) to reduce read power consumption. In a particular embodiment, a most frequently used address space can be mapped to these closer bundles, reducing read power by as much as 70%. As such, segmentation and allocation of frequently accessed addresses to specific physical locations may further reduce read power consumption.

Note that the segmentation shown in FIG. **6** is just one possible implementation of this scheme. Namely, FIG. **6** shows an arrangement of static logic **690** using two 4:1 multiplexers to combine static latch outputs from 8 bundles and ultimately using a 2:1 multiplexer to obtain one final read data. While this scheme may be suitable for delay considerations (as only two multiplexer stages are involved after the bundle-included SDL), inclusion of outputs from multiple SDLs can impact routing. If the bit slice is routing limited, an alternate design may multiplex an SDL output from a given bundle in a daisy chain fashion to the multiplexed input of the next bundle, until it reaches a final 2:1 multiplexer. In this scenario there may be additional multiplexer delays for timing purposes. In other cases, any hybrid design between the above two design alternatives may be used depending on which aspect (timing or layout) is of more interest. In embodiments, at least a portion of a readout path internal to a bundle is implemented as static logic. With this configuration, power consumption in the memory may be the same whether read operations are for a logic 1 or logic 0 value.

In the embodiment of FIG. **6**, there is a latch circuit such as an SDL in each segment (e.g., bundle) of the segmented memory array and the output of each segment is combined via a static multiplexer structure. With a configuration as in FIG. **6**, a major read Cdyn contributor, namely a global bitline, is instead implemented with a segmented global bitline that can use static logic resulting in lower read power.

As further illustrated in FIG. **6**, a memory may include a mapping circuit **685**. Mapping circuit **685** may be configured in different embodiments as hardware circuitry, software, firmware and/or combinations thereof. In operation, mapping circuit **685** may receive incoming memory access requests including write requests and, based at least in part on a priority of data associated with a given write request, re-map an address of the memory access request to another address within memory **600** that is associated with a lower power-consuming segment of the memory. For example, when an incoming memory access request is for data having a high priority, the request may be re-mapped to a given one of bundles $\mathbf{605}_3$ or $\mathbf{605}_4$, where these two bundles are the closest bundles to static logic **690**. To this end, mapping circuit **685** may include or be associated with a non-volatile storage such as a fuse storage that stores segment identification information. More specifically, this segment identification information can be used to identify given segments of the memory and characteristic information, such as corresponding power consumption, distance information, and/or priority association. In this way, based at least in part on priority associated with a given memory access request, mapping circuit **685** can identify an appropriate bundle **605** or other segment of memory **600** for storage of particular incoming data.

Embodiments may leverage a hybrid memory configuration (having domino and static logic within each bundle) to store information that is more frequently accessed into bundles that have shorter couplings to corresponding selection circuits within static logic **690**. As will be described herein, different manners of identifying such information for storage in these more closely located bundles can occur.

Referring now to FIG. **7**, shown is a flow diagram of a method in accordance with an embodiment of the present invention. More specifically, method **700** in FIG. **7** is a method for identifying more frequently accessed data so that it can be provided for storage in low order segments of a memory array as described herein. Method **700**, which may be performed by hardware circuitry, software, firmware and/or combinations thereof can be implemented in different manners in various embodiments. For example, in some cases a compiler-based technique may be used to identify information that is frequently accessed. In other cases, counter-based mechanisms or cache-based mechanisms may be used to identify more frequently accessed and/or recently accessed information to be stored in low order storage segments.

As illustrated, method **700** begins by receiving data to be stored in a memory array (block **710**). In cases where method **700** is performed within control logic of or associated with a memory array, the data may be received, e.g., in a memory interface. Next, control passes to block **720** where a read frequency of this data can be identified. Different manners of identifying read frequency can occur in different embodiments. For example, in some cases data may be received with an indication of an expected read frequency (e.g., one of multiple such levels of read frequency). In other cases, data may be associated with a particular priority level. For example, in some cases given threads may have a given priority associated therewith. Such priority information may be based on programmer-provided information such as an identification of one or more threads as high priority threads. In another case, this priority information may be based on quality of service (QoS)-type information or so forth. In some cases, this priority information may identify data as being one of multiple levels of priority. In some cases, there may be two levels of priority, namely normal priority and higher priority. In this case, the higher priority data items can be stored in segments of the memory that are closer to the merging static logic. In other cases there can be additional priority levels, for example, low, medium and high levels. In one such embodiment, only the highest level priority data can be stored in the segments closest to the merging static logic, in one embodiment.

In any event, control passes to block **730** where a bundle is selected for storage of the data based on this identified read frequency. In embodiments, mapping logic of the memory interface may perform this bundle selection. In cases where this incoming data is of a high priority or is

otherwise to be frequently accessed, the selection may be of a bundle that has a shortest routing to the static logic that merges multiple bundle outputs. Understand that while the terminology of "bundle" is used herein, any type of segmentation of the memory into different portions, where one or more portions are adapted in closer relation to static logic of the memory array can be implemented. In any event, the result of block **730** is a mapping or identification of a given bundle for storage of the data. Note that in cases where an incoming write request is directed to a particular storage location (e.g., a given one of multiple general-purpose registers (e.g., RO)), this bundle selection may include a re-mapping of the received write request location to a register associated with the selected bundle (and which may be a different register). Finally, control passes to block **740** where the data may be stored in the selected bundle. Understand while shown at this high level in the embodiment of FIG. **7**, many variations and alternatives are possible.

Power Components

FIG. **8** illustrates a block diagram of a switching regulator according to an embodiment. One or more switching regulators shown in FIG. **8** may be incorporated in various systems discussed herein to provide power to one or more Integrated Circuit (IC) chips. While a single phase of the current-parking switching regulator with a single inductor may be discussed with reference to FIG. **8**, one or more of the multiple phases of the current-parking switching regulator may be implemented with a split inductor. Furthermore, a combination of one or more current-parking switching regulators (with or without a split inductor) may be used with one or more conventional electric power conversion devices to provide power to the load (e.g., logic circuitry **814**).

More particularly, FIG. **8** illustrates a system **800** that includes a switching regulator (sometimes referred to as a current-parking switching regulator). The current-parking switching regulator may be a multi-phase switching regulator in various embodiments. The multi-phase control unit **802** is coupled to multiple phases, where each phase may include one or more upstream phases **804** and one or more downstream phases **806**. As shown, an electrical power source **808** is coupled to upstream control logic **810** (which provides a current control mechanisms in each upstream phase). More than one upstream control logic may be used in various implementations. Each upstream phase may include an inductor (not shown) that is coupled to a respective downstream phase. In an embodiment, the upstream phases may each include one or more inductors. The multi-phase control unit **802** may configure any active upstream control logic **810**, e.g., to generate a current through an inductor coupled between the upstream phases and the downstream phases. The downstream control logic **812** may be configured by the multi-phase control unit **802** to be ON, OFF, or switching to regulate the voltage level at the load (e.g., logic circuitry **814**). In turn, the downstream control logic **812** may be configured by the multi-phase control unit **802** to maintain the voltage level at the load within a range based at least in part on Vmin (minimum voltage) and Vmax (maximum voltage) values.

In one embodiment, an inductor (coupled between a downstream phase and a respective upstream phase) may be positioned outside of a semiconductor package **816** that includes the load **814**. Another inductor (not shown) may be positioned inside of the package **816**, e.g., to reduce parasitic capacitance. In one embodiment, the inductor inside the package **816** may be a planar air-core inductor that is coupled to the logic circuitry **814** via one or more switching logic which include planar Metal-Oxide Semiconductor Field-Effect Transistors (MOSFETs). Furthermore, one or more of the components discussed herein (e.g., with reference to FIGS. **8**, **9** and **10**, including, for example, L3 cache, upstream control logic, and/or downstream control logic) may be provided in substrate layer(s) (e.g., between semiconductor packages), on an integrated circuit die, or outside of a semiconductor package (e.g., on a Printed Circuit Board (PCB)) in various embodiments.

FIG. **9** is a block diagram of a system **900** including a streaming multiprocessor **902**, in accordance with one or more embodiments. The streaming multiprocessor may include 32 Single-Instruction, Multiple Thread (SIMT) lanes **904** that are capable of collectively issuing up to 32 instructions per clock cycle, e.g., one from each of 32 threads. More or less lanes may be present depending on the implementation such as 64, 128, 256, etc. The SIMT lanes **904** may in turn include one or more: Arithmetic Logic Units (ALUs) **906**, Special Function Units (SFUs) **908**, memory units (MEM) **910**, and/or texture units (TEX) **912**.

In some embodiments, one or more of ALU(s) **906** and/or TEX unit(s) **912** may be low energy or high capacity, e.g., such as discussed with reference to items **920** and **922**. For example, the system may map 100% of the register addresses for threads **0-30** to the low energy portion and 100% of the register addresses for threads **31-127** to the high capacity portion. As another example, the system may map 20% of each thread's registers to the low energy portion and to map 80% of each thread's registers to the high capacity portion. Moreover, the system may determine the number of entries allocated per thread based on runtime information.

As illustrated in FIG. **9**, the streaming multiprocessor **902** also include a register file **914**, a scheduler logic **916** (e.g., for scheduling threads or thread groups, or both), and shared memory **918**, e.g., local scratch storage. As discussed herein, a "thread group" refers to a plurality of threads that are grouped with ordered (e.g., sequential or consecutive) thread indexes. Generally, a register file refers to an array of registers accessed by components of a processor (including a graphics processor) such as those discussed herein. The register file **914** includes a low energy portion or structure **920** and a high capacity portion or structure **922**. The streaming multiprocessor **902** may be configured to address the register file **914** using a single logical namespace for both the low energy portion and the high capacity portion.

In some embodiments, the system may include a number of physical registers which can be shared by the simultaneously running threads on the system. This allows the system to use a single namespace to implement a flexible register mapping scheme. A compiler may then allocate register live ranges to register addresses, and the compiler may use a register allocation mechanism to minimize or reduce the number of registers used per thread. Multiple live ranges can be allocated to the same register address as long as the live ranges do not overlap in an embodiment. This allows for determination, e.g., at runtime and after instructions have been compiled, of how many entries per thread will be allocated in the low energy portion versus the high capacity portion. For example, the system may map 100% of the register addresses for threads **0-30** to the low energy portion and 100% of the register addresses for threads **31-127** to the high capacity portion. As another example, the system may map 20% of each thread's registers to the low energy portion and to map 80% of each thread's registers to the high capacity portion. The system may determine the number of entries allocated per thread based on runtime information,

e.g., regarding the number of thread groups executing and the marginal benefit from launching more thread groups or allocating a smaller number of thread groups more space in the low energy portion.

FIG. **10** illustrates a block diagram of a parallel processing system **1000**, according to one embodiment. System **1000** includes a Parallel Processing (Previously Presented) subsystem **1002** which in turn includes one or more Parallel Processing Units (PPUs) PPU-**0** through PPU-P. Each PPU is coupled to a local Parallel Processing (PP) memory (e.g., Mem-**0** through MEM-P, respectively). In some embodiments, the PP subsystem system **1002** may include P number of PPUs. PPU-**0** **1004** and parallel processing memories **1006** may be implemented using one or more integrated circuit devices, such as programmable processors, Application Specific Integrated Circuits (ASICs), or memory devices.

Referring to FIG. **10**, several optional switch or connections **1007** are shown that may be used in system **1000** to manage power. While several switches **1007** are shown, embodiments are not limited to the specifically shown switches and more or less switches may be utilized depending on the implementation. These connections/switches **1007** may be utilized for clock gating or general power gating. Hence, items **1007** may include one or more of a power transistor, on-die switch, power plane connections, or the like. In an embodiment, prior to shutting power to a portion of system **1000** via switches/connections **1007**, logic (e.g., a microcontroller, digital signal processor, firmware, etc.) may ensure the results of operation are committed (e.g., to memory) or finalized to maintain correctness.

Further, in some embodiments, one or more of PPUs in parallel processing subsystem **1002** are graphics processors with rendering pipelines that may be configured to perform various tasks such as those discussed herein with respect to other figures. The graphics information/data may be communicated via memory bridge **1008** with other components of a computing system (including components of system **1000**). The data may be communicated via a shared bus and/or one or more interconnect(s) **1010** (including, for example, one or more direct or point-to-point links). PPU-**0** **1004** may access its local parallel processing memory **1014** (which may be used as graphics memory including, e.g., a frame buffer) to store and update pixel data, delivering pixel data to a display device (such as those discussed herein), etc. In some embodiments, the parallel processing subsystem **1002** may include one or more PPUs that operate as graphics processors and one or more other PPUs that operate to perform general-purpose computations. The PPUs may be identical or different, and each PPU may have access to its own dedicated parallel processing memory device(s), no dedicated parallel processing memory device(s), or a shared memory device or cache.

In an embodiment, operations performed by PPUs may be controlled by another processor (or one of the PPUs) generally referred to as a master processor or processor core. In one embodiment, the master processor/core may write a stream of commands for each PPU to a push buffer in various locations such as a main system memory, a cache, or other memory such as those discussed herein with reference to other figures. The written commands may then be read by each PPU and executed asynchronously relative to the operation of master processor/core.

Furthermore, as shown in FIG. **10**, PPU-**0** includes a front end logic **1020** which may include an Input/Output (I/O or IO) unit (e.g., to communicate with other components of system **1000** through the memory bridge **1008**) and/or a host interface (e.g., which receives commands related to processing tasks). The front end **1020** may receive commands read by the host interface (for example from the push buffer)). The front end **1020** in turn provides the commands to a work scheduling unit **1022** that schedules and allocates operation(s)/task(s) associated with the commands to a processing cluster array or arithmetic subsystem **1024** for execution.

As shown in FIG. **10**, the processing cluster array **1024** may include one or more General Processing Cluster (GPC) units (e.g., GPC-**0** **1026**, GPC-**1** **1028**, through GPC-M **1030**). Each GPC may be capable of executing a large number (e.g., hundreds or thousands) of threads concurrently, where each thread is an instance of a program. In various applications, different GPCs may be allocated for processing different types of programs or for performing different types of computations. For example, in a graphics application, a first set of GPCs (e.g., including one or more GPC units) may be allocated to perform tessellation operations and to produce primitive topologies for patches, and a second set of GPCs (e.g., including one or more GPC units) may be allocated to perform tessellation shading to evaluate patch parameters for the primitive topologies and to determine vertex positions and other per-vertex attributes. The allocation of GPCs may vary depending on the workload arising for each type of program or computation.

Additionally, processing tasks that are assigned by the work scheduling unit **1022** may include indices of data to be processed, such surface/patch data, primitive data, vertex data, pixel data, and/or state parameters and commands defining how the data is to be processed (e.g., what program is to be executed). The work scheduling unit **1022** may be configured to fetch the indices corresponding to the tasks, or may receive the indices from front end **1020**. Front end **1020** may also ensure that GPCs are configured to a valid state before the processing specified by the push buffers is initiated.

In one embodiment, the communication path **1012** is a Peripheral Component Interface (PCI) express (or PCI-e) link, in which dedicated lanes may be allocated to each PPU. Other communication paths may also be used. For example, commands related to processing tasks may be directed to the host interface **1018**, while commands related to memory operations (e.g., reading from or writing to parallel processing memory **1014**) may be directed to a memory crossbar unit **1032**.

In some embodiments, parallel processing subsystem **1002** may be implemented as an add-in card that is inserted into an expansion slot of computer system or server (such as a blade server). In other embodiments, a PPU may be integrated on a single chip with a bus bridge, such as memory bridge **1008**, an I/O bridge, etc. In still other embodiments, some or all components of PPU may be integrated on a single integrated circuit chip with one or more other processor cores, memory devices, caches, etc.

Furthermore, one of the major problems with today's modern processors is they have hit a clock rate limit at around 4 GHz. At this point they just generate too much heat for the current technology and require special and expensive cooling solutions. This is because as we increase the clock rate, the power consumption rises. In fact, the power consumption of a CPU, if you fix the voltage, is approximately the cube of its clock rate. To make this worse, as you increase the heat generated by the CPU, for the same clock rate, the power consumption also increases due to the properties of the silicon. This conversion of power into heat is a complete waste of energy. This increasingly inefficient

use of power eventually means you are unable to either power or cool the processor sufficiently and you reach the thermal limits of the device or its housing, the so-called power wall.

Faced with not being able to increase the clock rate, making forever-faster processors, the processor manufacturers had to come up with another game plan. They have been forced down the route of adding more cores to processors, rather than continuously trying to increase CPU clock rates and/or extract more instructions per clock through instruction-level parallelism.

Moreover, power usage is a big consideration when designing machines that constantly run. Often the operating costs of running a supercomputer over just a few years can equate to the cost of installing it in the first place. Certainly, the cost of running such a machine over its lifetime will easily exceed the original installation costs. Power usage comes from the components themselves, but also from the cooling necessary to allow such computers to operate. Even one high-end workstation with four GPUs requires some planning on how to keep it cool. Unless you live in a cold climate and can banish the computer to somewhere cold, it will do a nice job of heating up the office for you. Put a number of such machines into one room, and very rapidly the air temperature in that room will start to rise to quite unacceptable levels.

A significant amount of power is therefore expended on installing air conditioning systems to ensure computers remain cool and can operate without producing errors. This is especially so where summer temperatures can reach 85 F/30 C or higher. Air conditioning is expensive to run. Significant thought should be given to how best to cool such a system and if the heat energy can in some way be reused. Liquid-cooled systems are very efficient in this way in that the liquid can be circulated through a heat exchanger and into a conventional heating system without any chance of the two liquids ever mixing. With the ever-increasing costs of natural resources, and the increasing pressures on companies to be seen as green, simply pumping the heat out the window is no longer economically or socially acceptable.

Liquid-cooled systems provide an interesting option in terms of recycling the waste heat energy. While an air-cooled system can only be used to heat the immediate area it is located in, heat from liquid-based coolants can be pumped elsewhere. By using a heat exchanger, the coolant can be cooled using conventional water. This can then be pumped into a heating system or even used to heat an outdoor swimming pool or other large body of water. Where a number of such systems are installed, such as in a company or university computer center, it can really make sense to use this waste heat energy to reduce the heating bill elsewhere in the organization.

Many supercomputer installations site themselves next to a major river precisely because they need a ready supply of cold water. Others use large cooling towers to dissipate the waste heat energy. Neither solution is particularly green. Having paid for the energy already it makes little sense to simply throw it away when it could so easily be used for heating. When considering power usage, we must also remember that program design actually plays a very big role in power consumption. The most expensive operation, power wise, is moving data on and off chip. Thus, simply making efficient use of the registers and shared memory within the device vastly reduces power usage. If you also consider that the total execution time for well-written programs is much smaller than for poorly written ones, you can see that rewriting old programs to make use of new features such as larger shared memory can even reduce operating costs in a large data center.

Referring to FIG. **10**, memory interface **1014** includes N partition units (e.g., Unit-**0** **1034**, Unit-**1** **1036**, through Unit-N **1038**) that are each directly coupled to a corresponding portion of parallel processing memory **1006** (such as Mem-**0** **1040**, Mem-**1** **1042**, through Mem-N **1044**). The number of partition units may generally be equal to the number of Previously Presented memory (or N as shown). The Previously Presented memory may be implemented with volatile memory such as Dynamic Random Access Memory (DRAM) or other types of volatile memory such as those discussed herein. In other embodiments, the number of partition units may not equal the number of memory devices. Graphics data (such as render targets, frame buffers, or texture maps) may be stored across Previously Presented memory devices, allowing partition units to write portions of graphics data in parallel to efficiently use the available bandwidth of the parallel processing memory **1006**.

Furthermore, any one of GPCs may process data to be written to any of the partition units within the parallel processing memory. Crossbar unit **1032** may be implemented as an interconnect that is configured to route the output of each GPC to the input of any partition unit or to another GPC for further processing. Hence, GPCs **1026** to **1030** may communicate with memory interface **1014** through crossbar unit **1032** to read from or write to various other (or external) memory devices. As shown, crossbar unit **1032** may directly communicate with the front end **1020**, as well as having a coupling (direct or indirect) to local memory **1006**, to allow the processing cores within the different GPCs to communicate with system memory and/or other memory that is not local to PPU. Furthermore, the crossbar unit **1032** may utilize virtual channels to organize traffic streams between the GPCs and partition units.

System Overview

FIG. **11** is a block diagram of a processing system **1100**, according to an embodiment. In various embodiments the system **1100** includes one or more processors **1102** and one or more graphics processors **1108**, and may be a single processor desktop system, a multiprocessor workstation system, or a server system having a large number of processors **1102** or processor cores **1107**. In one embodiment, the system **1100** is a processing platform incorporated within a system-on-a-chip (SoC) integrated circuit for use in mobile, handheld, or embedded devices.

An embodiment of system **1100** can include, or be incorporated within a server-based gaming platform, a game console, including a game and media console, a mobile gaming console, a handheld game console, or an online game console. In some embodiments system **1100** is a mobile phone, smart phone, tablet computing device or mobile Internet device. Data processing system **1100** can also include, couple with, or be integrated within a wearable device, such as a smart watch wearable device, smart eyewear device, augmented reality device, or virtual reality device. In some embodiments, data processing system **1100** is a television or set top box device having one or more processors **1102** and a graphical interface generated by one or more graphics processors **1108**.

In some embodiments, the one or more processors **1102** each include one or more processor cores **1107** to process instructions which, when executed, perform operations for system and user software. In some embodiments, each of the one or more processor cores **1107** is configured to process a specific instruction set **1109**. In some embodiments, instruc-

tion set **1109** may facilitate Complex Instruction Set Computing (CISC), Reduced Instruction Set Computing (RISC), or computing via a Very Long Instruction Word (VLIW). Multiple processor cores **1107** may each process a different instruction set **1109**, which may include instructions to facilitate the emulation of other instruction sets. Processor core **1107** may also include other processing devices, such a Digital Signal Processor (DSP).

In some embodiments, the processor **1102** includes cache memory **1104**. Depending on the architecture, the processor **1102** can have a single internal cache or multiple levels of internal cache. In some embodiments, the cache memory is shared among various components of the processor **1102**. In some embodiments, the processor **1102** also uses an external cache (e.g., a Level-3 (L3) cache or Last Level Cache (LLC)) (not shown), which may be shared among processor cores **1107** using known cache coherency techniques. A register file **1106** is additionally included in processor **1102** which may include different types of registers for storing different types of data (e.g., integer registers, floating point registers, status registers, and an instruction pointer register). Some registers may be general-purpose registers, while other registers may be specific to the design of the processor **1102**.

In some embodiments, processor **1102** is coupled with a processor bus **1110** to transmit communication signals such as address, data, or control signals between processor **1102** and other components in system **1100**. In one embodiment the system **1100** uses an exemplary 'hub' system architecture, including a memory controller hub **1116** and an Input Output (I/O) controller hub **1130**. A memory controller hub **1116** facilitates communication between a memory device and other components of system **1100**, while an I/O Controller Hub (ICH) **1130** provides connections to I/O devices via a local I/O bus. In one embodiment, the logic of the memory controller hub **1116** is integrated within the processor.

Memory device **1120** can be a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, flash memory device, phase-change memory device, or some other memory device having suitable performance to serve as process memory. In one embodiment the memory device **1120** can operate as system memory for the system **1100**, to store data **1122** and instructions **1121** for use when the one or more processors **1102** executes an application or process. Memory controller hub **1116** also couples with an optional external graphics processor **1112**, which may communicate with the one or more graphics processors **1108** in processors **1102** to perform graphics and media operations.

In some embodiments, ICH **1130** enables peripherals to connect to memory device **1120** and processor **1102** via a high-speed I/O bus. The I/O peripherals include, but are not limited to, an audio controller **1146**, a firmware interface **1128**, a wireless transceiver **1126** (e.g., Wi-Fi, Bluetooth), a data storage device **1124** (e.g., hard disk drive, flash memory, etc.), and a legacy I/O controller **1140** for coupling legacy (e.g., Personal System 2 (PS/2)) devices to the system. One or more Universal Serial Bus (USB) controllers **1142** connect input devices, such as keyboard and mouse **1144** combinations. A network controller **1134** may also couple with ICH **1130**. In some embodiments, a high-performance network controller (not shown) couples with processor bus **1110**. It will be appreciated that the system **1100** shown is exemplary and not limiting, as other types of data processing systems that are differently configured may also be used. For example, the I/O controller hub **1130** may be integrated within the one or more processor **1102**, or the memory controller hub **1116** and I/O controller hub **1130** may be integrated into a discreet external graphics processor, such as the external graphics processor **1112**.

FIG. **12** is a block diagram of an embodiment of a processor **1200** having one or more processor cores **1202**A-**1202**N, an integrated memory controller **1214**, and an integrated graphics processor **1208**. Those elements of FIG. **12** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such. Processor **1200** can include additional cores up to and including additional core **1202**N represented by the dashed lined boxes. Each of processor cores **1202**A-**1202**N includes one or more internal cache units **1204**A-**1204**N. In some embodiments each processor core also has access to one or more shared cached units **1206**.

The internal cache units **1204**A-**1204**N and shared cache units **1206** represent a cache memory hierarchy within the processor **1200**. The cache memory hierarchy may include at least one level of instruction and data cache within each processor core and one or more levels of shared mid-level cache, such as a Level 2 (L2), Level 3 (L3), Level 4 (L4), or other levels of cache, where the highest level of cache before external memory is classified as the LLC. In some embodiments, cache coherency logic maintains coherency between the various cache units **1206** and **1204**A-**1204**N.

In some embodiments, processor **1200** may also include a set of one or more bus controller units **1216** and a system agent core **1210**. The one or more bus controller units **1216** manage a set of peripheral buses, such as one or more Peripheral Component Interconnect buses (e.g., PCI, PCI Express). System agent core **1210** provides management functionality for the various processor components. In some embodiments, system agent core **1210** includes one or more integrated memory controllers **1214** to manage access to various external memory devices (not shown).

In some embodiments, one or more of the processor cores **1202**A-**1202**N include support for simultaneous multi-threading. In such embodiment, the system agent core **1210** includes components for coordinating and operating cores **1202**A-**1202**N during multi-threaded processing. System agent core **1210** may additionally include a power control unit (PCU), which includes logic and components to regulate the power state of processor cores **1202**A-**1202**N and graphics processor **1208**.

In some embodiments, processor **1200** additionally includes graphics processor **1208** to execute graphics processing operations. In some embodiments, the graphics processor **1208** couples with the set of shared cache units **1206**, and the system agent core **1210**, including the one or more integrated memory controllers **1214**. In some embodiments, a display controller **1211** is coupled with the graphics processor **1208** to drive graphics processor output to one or more coupled displays. In some embodiments, display controller **1211** may be a separate module coupled with the graphics processor via at least one interconnect, or may be integrated within the graphics processor **1208** or system agent core **1210**.

In some embodiments, a ring based interconnect unit **1212** is used to couple the internal components of the processor **1200**. However, an alternative interconnect unit may be used, such as a point-to-point interconnect, a switched interconnect, or other techniques, including tech-

niques well known in the art. In some embodiments, graphics processor **1208** couples with the ring interconnect **1212** via an I/O link **1213**.

The exemplary I/O link **1213** represents at least one of multiple varieties of I/O interconnects, including an on package I/O interconnect which facilitates communication between various processor components and a high-performance embedded memory module **1218**, such as an eDRAM module. In some embodiments, each of the processor cores **1202A-1202N** and graphics processor **1208** use embedded memory modules **1218** as a shared Last Level Cache.

In some embodiments, processor cores **1202A-1202N** are homogenous cores executing the same instruction set architecture. In another embodiment, processor cores **1202A-1202N** are heterogeneous in terms of instruction set architecture (ISA), where one or more of processor cores **1202A-1202N** execute a first instruction set, while at least one of the other cores executes a subset of the first instruction set or a different instruction set. In one embodiment processor cores **1202A-1202N** are heterogeneous in terms of microarchitecture, where one or more cores having a relatively higher power consumption couple with one or more power cores having a lower power consumption. Additionally, processor **1200** can be implemented on one or more chips or as an SoC integrated circuit having the illustrated components, in addition to other components.

FIG. **13** is a block diagram of a graphics processor **1300**, which may be a discrete graphics processing unit, or may be a graphics processor integrated with a plurality of processing cores. In some embodiments, the graphics processor communicates via a memory mapped I/O interface to registers on the graphics processor and with commands placed into the processor memory. In some embodiments, graphics processor **1300** includes a memory interface **1314** to access memory. Memory interface **1314** can be an interface to local memory, one or more internal caches, one or more shared external caches, and/or to system memory.

In some embodiments, graphics processor **1300** also includes a display controller **1302** to drive display output data to a display device **1320**. Display controller **1302** includes hardware for one or more overlay planes for the display and composition of multiple layers of video or user interface elements. In some embodiments, graphics processor **1300** includes a video codec engine **1306** to encode, decode, or transcode media to, from, or between one or more media encoding formats, including, but not limited to Moving Picture Experts Group (MPEG) formats such as MPEG-2, Advanced Video Coding (AVC) formats such as H.264/MPEG-4 AVC, as well as the Society of Motion Picture & Television Engineers (SMPTE) 421M/VC-1, and Joint Photographic Experts Group (JPEG) formats such as JPEG, and Motion JPEG (MJPEG) formats.

In some embodiments, graphics processor **1300** includes a block image transfer (BLIT) engine **1304** to perform two-dimensional (2D) rasterizer operations including, for example, bit-boundary block transfers. However, in one embodiment, 2D graphics operations are performed using one or more components of graphics processing engine (GPE) **1310**. In some embodiments, GPE **1310** is a compute engine for performing graphics operations, including three-dimensional (3D) graphics operations and media operations.

In some embodiments, GPE **1310** includes a 3D pipeline **1312** for performing 3D operations, such as rendering three-dimensional images and scenes using processing functions that act upon 3D primitive shapes (e.g., rectangle, triangle, etc.). The 3D pipeline **1312** includes programmable and fixed function elements that perform various tasks within the element and/or spawn execution threads to a 3D/Media sub-system **1315**. While 3D pipeline **1312** can be used to perform media operations, an embodiment of GPE **1310** also includes a media pipeline **1316** that is specifically used to perform media operations, such as video post-processing and image enhancement.

In some embodiments, media pipeline **1316** includes fixed function or programmable logic units to perform one or more specialized media operations, such as video decode acceleration, video de-interlacing, and video encode acceleration in place of, or on behalf of video codec engine **1306**. In some embodiments, media pipeline **1316** additionally includes a thread spawning unit to spawn threads for execution on 3D/Media sub-system **1315**. The spawned threads perform computations for the media operations on one or more graphics execution units included in 3D/Media sub-system **1315**.

In some embodiments, 3D/Media subsystem **1315** includes logic for executing threads spawned by 3D pipeline **1312** and media pipeline **1316**. In one embodiment, the pipelines send thread execution requests to 3D/Media subsystem **1315**, which includes thread dispatch logic for arbitrating and dispatching the various requests to available thread execution resources. The execution resources include an array of graphics execution units to process the 3D and media threads. In some embodiments, 3D/Media subsystem **1315** includes one or more internal caches for thread instructions and data. In some embodiments, the subsystem also includes shared memory, including registers and addressable memory, to share data between threads and to store output data.

Graphics Processing Engine

FIG. **14** is a block diagram of a graphics processing engine **1410** of a graphics processor in accordance with some embodiments. In one embodiment, the graphics processing engine (GPE) **1410** is a version of the GPE **1310** shown in FIG. **13**. Elements of FIG. **14** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such. For example, the 3D pipeline **1312** and media pipeline **1316** of FIG. **13** are illustrated. The media pipeline **1416** is optional in some embodiments of the GPE **1410** and may not be explicitly included within the GPE **1410**. For example and in at least one embodiment, a separate media and/or image processor is coupled to the GPE **1410**.

In some embodiments, GPE **1410** couples with or includes a command streamer **1403**, which provides a command stream to the 3D pipeline **1412** and/or media pipelines **1416**. In some embodiments, command streamer **1403** is coupled with memory, which can be system memory, or one or more of internal cache memory and shared cache memory. In some embodiments, command streamer **1403** receives commands from the memory and sends the commands to 3D pipeline **1412** and/or media pipeline **1416**. The commands are directives fetched from a ring buffer, which stores commands for the 3D pipeline **1412** and media pipeline **1416**. In one embodiment, the ring buffer can additionally include batch command buffers storing batches of multiple commands. The commands for the 3D pipeline **1412** can also include references to data stored in memory, such as but not limited to vertex and geometry data for the 3D pipeline **1412** and/or image data and memory objects for the media pipeline **1416**. The 3D pipeline **1412** and media pipeline **1416** process the commands and data by performing

operations via logic within the respective pipelines or by dispatching one or more execution threads to a graphics core array **1414**.

In various embodiments the 3D pipeline **1412** can execute one or more shader programs, such as vertex shaders, geometry shaders, pixel shaders, fragment shaders, compute shaders, or other shader programs, by processing the instructions and dispatching execution threads to the graphics core array **1414**. The graphics core array **1414** provides a unified block of execution resources. Multi-purpose execution logic (e.g., execution units) within the graphic core array **1414** includes support for various 3D API shader languages and can execute multiple simultaneous execution threads associated with multiple shaders.

In some embodiments the graphics core array **1414** also includes execution logic to perform media functions, such as video and/or image processing. In one embodiment, the execution units additionally include general-purpose logic that is programmable to perform parallel general purpose computational operations, in addition to graphics processing operations. The general purpose logic can perform processing operations in parallel or in conjunction with general purpose logic within the processor core(s) **1107** of FIG. **11** or core **1202**A-**1202**N as in FIG. **12**.

Output data generated by threads executing on the graphics core array **1414** can output data to memory in a unified return buffer (URB) **1418**. The URB **1418** can store data for multiple threads. In some embodiments the URB **1418** may be used to send data between different threads executing on the graphics core array **1414**. In some embodiments the URB **1418** may additionally be used for synchronization between threads on the graphics core array and fixed function logic within the shared function logic **1420**.

In some embodiments, graphics core array **1414** is scalable, such that the array includes a variable number of graphics cores, each having a variable number of execution units based on the target power and performance level of GPE **1410**. In one embodiment the execution resources are dynamically scalable, such that execution resources may be enabled or disabled as needed.

The graphics core array **1414** couples with shared function logic **1420** that includes multiple resources that are shared between the graphics cores in the graphics core array. The shared functions within the shared function logic **1420** are hardware logic units that provide specialized supplemental functionality to the graphics core array **1414**. In various embodiments, shared function logic **1420** includes but is not limited to sampler **1421**, math **1422**, and inter-thread communication (ITC) **1423** logic. Additionally, some embodiments implement one or more cache(s) **1425** within the shared function logic **1420**. A shared function is implemented where the demand for a given specialized function is insufficient for inclusion within the graphics core array **1414**. Instead a single instantiation of that specialized function is implemented as a stand-alone entity in the shared function logic **1420** and shared among the execution resources within the graphics core array **1414**. The precise set of functions that are shared between the graphics core array **1414** and included within the graphics core array **1414** varies between embodiments.

FIG. **15** is a block diagram of another embodiment of a graphics processor **1500**. Elements of FIG. **15** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

In some embodiments, graphics processor **1500** includes a ring interconnect **1502**, a pipeline front-end **1504**, a media engine **1537**, and graphics cores **1580**A-**1580**N. In some embodiments, ring interconnect **1502** couples the graphics processor to other processing units, including other graphics processors or one or more general-purpose processor cores. In some embodiments, the graphics processor is one of many processors integrated within a multi-core processing system.

In some embodiments, graphics processor **1500** receives batches of commands via ring interconnect **1502**. The incoming commands are interpreted by a command streamer **1503** in the pipeline front-end **1504**. In some embodiments, graphics processor **1500** includes scalable execution logic to perform 3D geometry processing and media processing via the graphics core(s) **1580**A-**1580**N. For 3D geometry processing commands, command streamer **1503** supplies commands to geometry pipeline **1536**. For at least some media processing commands, command streamer **1503** supplies the commands to a video front end **1534**, which couples with a media engine **1537**. In some embodiments, media engine **1537** includes a Video Quality Engine (VQE) **1530** for video and image post-processing and a multi-format encode/decode (MFX) **1533** engine to provide hardware-accelerated media data encode and decode. In some embodiments, geometry pipeline **1536** and media engine **1537** each generate execution threads for the thread execution resources provided by at least one graphics core **1580**A.

In some embodiments, graphics processor **1500** includes scalable thread execution resources featuring modular cores **1580**A-**1580**N (sometimes referred to as core slices), each having multiple sub-cores **1550**A-**1550**N, **1560**A-**1560**N (sometimes referred to as core sub-slices). In some embodiments, graphics processor **1500** can have any number of graphics cores **1580**A through **1580**N. In some embodiments, graphics processor **1500** includes a graphics core **1580**A having at least a first sub-core **1550**A and a second sub-core **1560**A. In other embodiments, the graphics processor is a low power processor with a single sub-core (e.g., **1550**A). In some embodiments, graphics processor **1500** includes multiple graphics cores **1580**A-**1580**N, each including a set of first sub-cores **1550**A-**1550**N and a set of second sub-cores **1560**A-**1560**N. Each sub-core in the set of first sub-cores **1550**A-**1550**N includes at least a first set of execution units **1552**A-**1552**N and media/texture samplers **1554**A-**1554**N. Each sub-core in the set of second sub-cores **1560**A-**1560**N includes at least a second set of execution units **1562**A-**1562**N and samplers **1564**A-**1564**N. In some embodiments, each sub-core **1550**A-**1550**N, **1560**A-**1560**N shares a set of shared resources **1570**A-**1570**N. In some embodiments, the shared resources include shared cache memory and pixel operation logic. Other shared resources may also be included in the various embodiments of the graphics processor.

Execution Units

FIG. **16** illustrates thread execution logic **1600** including an array of processing elements employed in some embodiments of a GPE. Elements of FIG. **16** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

In some embodiments, thread execution logic **1600** includes a shader processor **1602**, a thread dispatcher **1604**, instruction cache **1606**, a scalable execution unit array including a plurality of execution units **1608**A-**1608**N, a sampler **1610**, a data cache **1612**, and a data port **1614**. In one embodiment the scalable execution unit array can

dynamically scale by enabling or disabling one or more execution units (e.g., any of execution unit **1608**A, **1608**B, **1608**C, **1608**D, through **1608**N-**1** and **1608**N) based on the computational requirements of a workload. In one embodiment the included components are interconnected via an interconnect fabric that links to each of the components. In some embodiments, thread execution logic **1600** includes one or more connections to memory, such as system memory or cache memory, through one or more of instruction cache **1606**, data port **1614**, sampler **1610**, and execution units **1608**A-**1608**N. In some embodiments, each execution unit (e.g. **1608**A) is a stand-alone programmable general purpose computational unit that is capable of executing multiple simultaneous hardware threads while processing multiple data elements in parallel for each thread. In various embodiments, the array of execution units **1608**A-**1608**N is scalable to include any number individual execution units.

In some embodiments, the execution units **1608**A-**1608**N are primarily used to execute shader programs. A shader processor **1602** can process the various shader programs and dispatch execution threads associated with the shader programs via a thread dispatcher **1604**. In one embodiment the thread dispatcher includes logic to arbitrate thread initiation requests from the graphics and media pipelines and instantiate the requested threads on one or more execution unit in the execution units **1608**A-**1608**N. For example, the geometry pipeline (e.g., **1536** of FIG. **15**) can dispatch vertex, tessellation, or geometry shaders to the thread execution logic **1600** (FIG. **16**) for processing. In some embodiments, thread dispatcher **1604** can also process runtime thread spawning requests from the executing shader programs.

In some embodiments, the execution units **1608**A-**1608**N support an instruction set that includes native support for many standard 3D graphics shader instructions, such that shader programs from graphics libraries (e.g., Direct 3D and OpenGL) are executed with a minimal translation. The execution units support vertex and geometry processing (e.g., vertex programs, geometry programs, vertex shaders), pixel processing (e.g., pixel shaders, fragment shaders) and general-purpose processing (e.g., compute and media shaders). Each of the execution units **1608**A-**1608**N is capable of multi-issue single instruction multiple data (SIMD) execution and multi-threaded operation enables an efficient execution environment in the face of higher latency memory accesses. Each hardware thread within each execution unit has a dedicated high-bandwidth register file and associated independent thread-state. Execution is multi-issue per clock to pipelines capable of integer, single and double precision floating point operations, SIMD branch capability, logical operations, transcendental operations, and other miscellaneous operations. While waiting for data from memory or one of the shared functions, dependency logic within the execution units **1608**A-**1608**N causes a waiting thread to sleep until the requested data has been returned. While the waiting thread is sleeping, hardware resources may be devoted to processing other threads. For example, during a delay associated with a vertex shader operation, an execution unit can perform operations for a pixel shader, fragment shader, or another type of shader program, including a different vertex shader.

Each execution unit in execution units **1608**A-**1608**N operates on arrays of data elements. The number of data elements is the "execution size," or the number of channels for the instruction. An execution channel is a logical unit of execution for data element access, masking, and flow control within instructions. The number of channels may be independent of the number of physical Arithmetic Logic Units (ALUs) or Floating Point Units (FPUs) for a particular graphics processor. In some embodiments, execution units **1608**A-**1608**N support integer and floating-point data types.

The execution unit instruction set includes SIMD instructions. The various data elements can be stored as a packed data type in a register and the execution unit will process the various elements based on the data size of the elements. For example, when operating on a 256-bit wide vector, the 256 bits of the vector are stored in a register and the execution unit operates on the vector as four separate 64-bit packed data elements (Quad-Word (QW) size data elements), eight separate 32-bit packed data elements (Double Word (DW) size data elements), sixteen separate 16-bit packed data elements (Word (W) size data elements), or thirty-two separate 8-bit data elements (byte (B) size data elements). However, different vector widths and register sizes are possible.

One or more internal instruction caches (e.g., 1606) are included in the thread execution logic **1600** to cache thread instructions for the execution units. In some embodiments, one or more data caches (e.g., 1612) are included to cache thread data during thread execution. In some embodiments, a sampler **1610** is included to provide texture sampling for 3D operations and media sampling for media operations. In some embodiments, sampler **1610** includes specialized texture or media sampling functionality to process texture or media data during the sampling process before providing the sampled data to an execution unit.

During execution, the graphics and media pipelines send thread initiation requests to thread execution logic **1600** via thread spawning and dispatch logic. Once a group of geometric objects has been processed and rasterized into pixel data, pixel processor logic (e.g., pixel shader logic, fragment shader logic, etc.) within the shader processor **1602** is invoked to further compute output information and cause results to be written to output surfaces (e.g., color buffers, depth buffers, stencil buffers, etc.). In some embodiments, a pixel shader or fragment shader calculates the values of the various vertex attributes that are to be interpolated across the rasterized object. In some embodiments, pixel processor logic within the shader processor **1602** then executes an application programming interface (API)-supplied pixel or fragment shader program. To execute the shader program, the shader processor **1602** dispatches threads to an execution unit (e.g., **1608**A) via thread dispatcher **1604**. In some embodiments, pixel shader **1602** uses texture sampling logic in the sampler **1610** to access texture data in texture maps stored in memory. Arithmetic operations on the texture data and the input geometry data compute pixel color data for each geometric fragment, or discards one or more pixels from further processing.

In some embodiments, the data port **1614** provides a memory access mechanism for the thread execution logic **1600** output processed data to memory for processing on a graphics processor output pipeline. In some embodiments, the data port **1614** includes or couples to one or more cache memories (e.g., data cache **1612**) to cache data for memory access via the data port.

FIG. **17** is a block diagram illustrating a graphics processor instruction formats **1700** according to some embodiments. In one or more embodiment, the graphics processor execution units support an instruction set having instructions in multiple formats. The solid lined boxes illustrate the components that are generally included in an execution unit instruction, while the dashed lines include components that are optional or that are only included in a sub-set of the instructions. In some embodiments, instruction format **1700**

described and illustrated are macro-instructions, in that they are instructions supplied to the execution unit, as opposed to micro-operations resulting from instruction decode once the instruction is processed.

In some embodiments, the graphics processor execution units natively support instructions in a 128-bit instruction format **1710**. A 64-bit compacted instruction format **1730** is available for some instructions based on the selected instruction, instruction options, and number of operands. The native 128-bit instruction format **1710** provides access to all instruction options, while some options and operations are restricted in the 64-bit format **1730**. The native instructions available in the 64-bit format **1730** vary by embodiment. In some embodiments, the instruction is compacted in part using a set of index values in an index field **1713**. The execution unit hardware references a set of compaction tables based on the index values and uses the compaction table outputs to reconstruct a native instruction in the 128-bit instruction format **1710**.

For each format, instruction opcode **1712** defines the operation that the execution unit is to perform. The execution units execute each instruction in parallel across the multiple data elements of each operand. For example, in response to an add instruction the execution unit performs a simultaneous add operation across each color channel representing a texture element or picture element. By default, the execution unit performs each instruction across all data channels of the operands. In some embodiments, instruction control field **1714** enables control over certain execution options, such as channels selection (e.g., predication) and data channel order (e.g., swizzle). For instructions in the 128-bit instruction format **1710** an exec-size field **1716** limits the number of data channels that will be executed in parallel. In some embodiments, exec-size field **1716** is not available for use in the 64-bit compact instruction format **1730**.

Some execution unit instructions have up to three operands including two source operands, src**0** **1720**, src**1** **1722**, and one destination **1718**. In some embodiments, the execution units support dual destination instructions, where one of the destinations is implied. Data manipulation instructions can have a third source operand (e.g., SRC**2** **1724**), where the instruction opcode **1712** determines the number of source operands. An instruction's last source operand can be an immediate (e.g., hard-coded) value passed with the instruction.

In some embodiments, the 128-bit instruction format **1710** includes an access/address mode field **1726** specifying, for example, whether direct register addressing mode or indirect register addressing mode is used. When direct register addressing mode is used, the register address of one or more operands is directly provided by bits in the instruction.

In some embodiments, the 128-bit instruction format **1710** includes an access/address mode field **1726**, which specifies an address mode and/or an access mode for the instruction. In one embodiment the access mode is used to define a data access alignment for the instruction. Some embodiments support access modes including a 16-byte aligned access mode and a 1-byte aligned access mode, where the byte alignment of the access mode determines the access alignment of the instruction operands. For example, when in a first mode, the instruction may use byte-aligned addressing for source and destination operands and when in a second mode, the instruction may use 16-byte-aligned addressing for all source and destination operands.

In one embodiment, the address mode portion of the access/address mode field **1726** determines whether the instruction is to use direct or indirect addressing. When direct register addressing mode is used bits in the instruction directly provide the register address of one or more operands. When indirect register addressing mode is used, the register address of one or more operands may be computed based on an address register value and an address immediate field in the instruction.

In some embodiments instructions are grouped based on opcode **1712** bit-fields to simplify Opcode decode **1740**. For an 8-bit opcode, bits **4**, **5**, and **6** allow the execution unit to determine the type of opcode. The precise opcode grouping shown is merely an example. In some embodiments, a move and logic opcode group **1742** includes data movement and logic instructions (e.g., move (mov), compare (cmp)). In some embodiments, move and logic group **1742** shares the five most significant bits (MSB), where move (mov) instructions are in the form of 0000xxxxb and logic instructions are in the form of 0001xxxxb. A flow control instruction group **1744** (e.g., call, jump (jmp)) includes instructions in the form of 0010xxxxb (e.g., 0x20). A miscellaneous instruction group **1746** includes a mix of instructions, including synchronization instructions (e.g., wait, send) in the form of 0011xxxxb (e.g., 0x30). A parallel math instruction group **1748** includes component-wise arithmetic instructions (e.g., add, multiply (mul)) in the form of 0100xxxxb (e.g., 0x40). The parallel math group **1748** performs the arithmetic operations in parallel across data channels. The vector math group **1750** includes arithmetic instructions (e.g., dp4) in the form of 0101xxxxb (e.g., 0x50). The vector math group performs arithmetic such as dot product calculations on vector operands.

Graphics Pipeline

FIG. **18** is a block diagram of another embodiment of a graphics processor **1800**. Elements of FIG. **18** having the same reference numbers (or names) as the elements of any other figure herein can operate or function in any manner similar to that described elsewhere herein, but are not limited to such.

In some embodiments, graphics processor **1800** includes a graphics pipeline **1820**, a media pipeline **1830**, a display engine **1840**, thread execution logic **1850**, and a render output pipeline **1870**. In some embodiments, graphics processor **1800** is a graphics processor within a multi-core processing system that includes one or more general purpose processing cores. The graphics processor is controlled by register writes to one or more control registers (not shown) or via commands issued to graphics processor **1800** via a ring interconnect **1802**. In some embodiments, ring interconnect **1802** couples graphics processor **1800** to other processing components, such as other graphics processors or general-purpose processors. Commands from ring interconnect **1802** are interpreted by a command streamer **1803**, which supplies instructions to individual components of graphics pipeline **1820** or media pipeline **1830**.

In some embodiments, command streamer **1803** directs the operation of a vertex fetcher **1805** that reads vertex data from memory and executes vertex-processing commands provided by command streamer **1803**. In some embodiments, vertex fetcher **1805** provides vertex data to a vertex shader **1807**, which performs coordinate space transformation and lighting operations to each vertex. In some embodiments, vertex fetcher **1805** and vertex shader **1807** execute vertex-processing instructions by dispatching execution threads to execution units **1852**A-**1852**B via a thread dispatcher **1831**.

In some embodiments, execution units **1852**A-**1852**B are an array of vector processors having an instruction set for performing graphics and media operations. In some embodiments, execution units **1852**A-**1852**B have an attached L1 cache **1851** that is specific for each array or shared between the arrays. The cache can be configured as a data cache, an instruction cache, or a single cache that is partitioned to contain data and instructions in different partitions.

In some embodiments, graphics pipeline **1820** includes tessellation components to perform hardware-accelerated tessellation of 3D objects. In some embodiments, a programmable hull shader **1811** configures the tessellation operations. A programmable domain shader **1817** provides back-end evaluation of tessellation output. A tessellator **1813** operates at the direction of hull shader **1811** and contains special purpose logic to generate a set of detailed geometric objects based on a coarse geometric model that is provided as input to graphics pipeline **1820**. In some embodiments, if tessellation is not used, tessellation components (e.g., hull shader **1811**, tessellator **1813**, and domain shader **1817**) can be bypassed.

In some embodiments, complete geometric objects can be processed by a geometry shader **1819** via one or more threads dispatched to execution units **1852**A-**1852**B, or can proceed directly to the clipper **1829**. In some embodiments, the geometry shader operates on entire geometric objects, rather than vertices or patches of vertices as in previous stages of the graphics pipeline. If the tessellation is disabled the geometry shader **1819** receives input from the vertex shader **1807**. In some embodiments, geometry shader **1819** is programmable by a geometry shader program to perform geometry tessellation if the tessellation units are disabled.

Before rasterization, a clipper **1829** processes vertex data. The clipper **1829** may be a fixed function clipper or a programmable clipper having clipping and geometry shader functions. In some embodiments, a rasterizer and depth test component **1873** in the render output pipeline **1870** dispatches pixel shaders to convert the geometric objects into their per pixel representations. In some embodiments, pixel shader logic is included in thread execution logic **1850**. In some embodiments, an application can bypass the rasterizer and depth test component **1873** and access un-rasterized vertex data via a stream out unit **1823**.

The graphics processor **1800** has an interconnect bus, interconnect fabric, or some other interconnect mechanism that allows data and message passing amongst the major components of the processor. In some embodiments, execution units **1852**A-**1852**B and associated cache(s) **1851**, texture and media sampler **1854**, and texture/sampler cache **1858** interconnect via a data port **1856** to perform memory access and communicate with render output pipeline components of the processor. In some embodiments, sampler **1854**, caches **1851**, **1858** and execution units **1852**A-**1852**B each have separate memory access paths.

In some embodiments, render output pipeline **1870** contains a rasterizer and depth test component **1873** that converts vertex-based objects into an associated pixel-based representation. In some embodiments, the rasterizer logic includes a windower/masker unit to perform fixed function triangle and line rasterization. An associated render cache **1878** and depth cache **1879** are also available in some embodiments. A pixel operations component **1877** performs pixel-based operations on the data, though in some instances, pixel operations associated with 2D operations (e.g. bit block image transfers with blending) are performed by the 2D engine **1841**, or substituted at display time by the display controller **1843** using overlay display planes. In some embodiments, a shared L3 cache **1875** is available to all graphics components, allowing the sharing of data without the use of main system memory.

In some embodiments, graphics processor media pipeline **1830** includes a media engine **1837** and a video front end **1834**. In some embodiments, video front end **1834** receives pipeline commands from the command streamer **1803**. In some embodiments, media pipeline **1830** includes a separate command streamer. In some embodiments, video front-end **1834** processes media commands before sending the command to the media engine **1837**. In some embodiments, media engine **1837** includes thread spawning functionality to spawn threads for dispatch to thread execution logic **1850** via thread dispatcher **1831**.

In some embodiments, graphics processor **1800** includes a display engine **1840**. In some embodiments, display engine **1840** is external to processor **1800** and couples with the graphics processor via the ring interconnect **1802**, or some other interconnect bus or fabric. In some embodiments, display engine **1840** includes a 2D engine **1841** and a display controller **1843**. In some embodiments, display engine **1840** contains special purpose logic capable of operating independently of the 3D pipeline. In some embodiments, display controller **1843** couples with a display device (not shown), which may be a system integrated display device, as in a laptop computer, or an external display device attached via a display device connector.

In some embodiments, graphics pipeline **1820** and media pipeline **1830** are configurable to perform operations based on multiple graphics and media programming interfaces and are not specific to any one application programming interface (API). In some embodiments, driver software for the graphics processor translates API calls that are specific to a particular graphics or media library into commands that can be processed by the graphics processor. In some embodiments, support is provided for the Open Graphics Library (OpenGL), Open Computing Language (OpenCL), and/or Vulkan graphics and compute API, all from the Khronos Group. In some embodiments, support may also be provided for the Direct3D library from the Microsoft Corporation. In some embodiments, a combination of these libraries may be supported. Support may also be provided for the Open Source Computer Vision Library (OpenCV). A future API with a compatible 3D pipeline would also be supported if a mapping can be made from the pipeline of the future API to the pipeline of the graphics processor.

Graphics Pipeline Programming

FIG. **19**A is a block diagram illustrating a graphics processor command format **1900** according to some embodiments. FIG. **19**B is a block diagram illustrating a graphics processor command sequence **1910** according to an embodiment. The solid lined boxes in FIG. **19**A illustrate the components that are generally included in a graphics command while the dashed lines include components that are optional or that are only included in a sub-set of the graphics commands. The exemplary graphics processor command format **1900** of FIG. **19**A includes data fields to identify a target client **1902** of the command, a command operation code (opcode) **1904**, and the relevant data **1906** for the command. A sub-opcode **1905** and a command size **1908** are also included in some commands.

In some embodiments, client **1902** specifies the client unit of the graphics device that processes the command data. In some embodiments, a graphics processor command parser examines the client field of each command to condition the further processing of the command and route the command data to the appropriate client unit. In some embodiments, the

graphics processor client units include a memory interface unit, a render unit, a 2D unit, a 3D unit, and a media unit. Each client unit has a corresponding processing pipeline that processes the commands. Once the command is received by the client unit, the client unit reads the opcode **1904** and, if present, sub-opcode **1905** to determine the operation to perform. The client unit performs the command using information in data field **1906**. For some commands an explicit command size **1908** is expected to specify the size of the command. In some embodiments, the command parser automatically determines the size of at least some of the commands based on the command opcode. In some embodiments commands are aligned via multiples of a double word.

The flow diagram in FIG. **19**B shows an exemplary graphics processor command sequence **1910**. In some embodiments, software or firmware of a data processing system that features an embodiment of a graphics processor uses a version of the command sequence shown to set up, execute, and terminate a set of graphics operations. A sample command sequence is shown and described for purposes of example only as embodiments are not limited to these specific commands or to this command sequence. Moreover, the commands may be issued as batch of commands in a command sequence, such that the graphics processor will process the sequence of commands in at least partially concurrence.

In some embodiments, the graphics processor command sequence **1910** may begin with a pipeline flush command **1912** to cause any active graphics pipeline to complete the currently pending commands for the pipeline. In some embodiments, the 3D pipeline **1922** and the media pipeline **1924** do not operate concurrently. The pipeline flush is performed to cause the active graphics pipeline to complete any pending commands. In response to a pipeline flush, the command parser for the graphics processor will pause command processing until the active drawing engines complete pending operations and the relevant read caches are invalidated. Optionally, any data in the render cache that is marked 'dirty' can be flushed to memory. In some embodiments, pipeline flush command **1912** can be used for pipeline synchronization or before placing the graphics processor into a low power state.

In some embodiments, a pipeline select command **1913** is used when a command sequence requires the graphics processor to explicitly switch between pipelines. In some embodiments, a pipeline select command **1913** is required only once within an execution context before issuing pipeline commands unless the context is to issue commands for both pipelines. In some embodiments, a pipeline flush command **1912** is required immediately before a pipeline switch via the pipeline select command **1913**.

In some embodiments, a pipeline control command **1914** configures a graphics pipeline for operation and is used to program the 3D pipeline **1922** and the media pipeline **1924**. In some embodiments, pipeline control command **1914** configures the pipeline state for the active pipeline. In one embodiment, the pipeline control command **1914** is used for pipeline synchronization and to clear data from one or more cache memories within the active pipeline before processing a batch of commands.

In some embodiments, return buffer state commands **1916** are used to configure a set of return buffers for the respective pipelines to write data. Some pipeline operations require the allocation, selection, or configuration of one or more return buffers into which the operations write intermediate data during processing. In some embodiments, the graphics processor also uses one or more return buffers to store output data and to perform cross thread communication. In some embodiments, the return buffer state **1916** includes selecting the size and number of return buffers to use for a set of pipeline operations.

The remaining commands in the command sequence differ based on the active pipeline for operations. Based on a pipeline determination **1920**, the command sequence is tailored to the 3D pipeline **1922** beginning with the 3D pipeline state **1930** or the media pipeline **1924** beginning at the media pipeline state **1940**.

The commands to configure the 3D pipeline state **1930** include 3D state setting commands for vertex buffer state, vertex element state, constant color state, depth buffer state, and other state variables that are to be configured before 3D primitive commands are processed. The values of these commands are determined at least in part based on the particular 3D API in use. In some embodiments, 3D pipeline state **1930** commands are also able to selectively disable or bypass certain pipeline elements if those elements will not be used.

In some embodiments, 3D primitive **1932** command is used to submit 3D primitives to be processed by the 3D pipeline. Commands and associated parameters that are passed to the graphics processor via the 3D primitive **1932** command are forwarded to the vertex fetch function in the graphics pipeline. The vertex fetch function uses the 3D primitive **1932** command data to generate vertex data structures. The vertex data structures are stored in one or more return buffers. In some embodiments, 3D primitive **1932** command is used to perform vertex operations on 3D primitives via vertex shaders. To process vertex shaders, 3D pipeline **1922** dispatches shader execution threads to graphics processor execution units.

In some embodiments, 3D pipeline **1922** is triggered via an execute **1934** command or event. In some embodiments, a register write triggers command execution. In some embodiments execution is triggered via a 'go' or 'kick' command in the command sequence. In one embodiment, command execution is triggered using a pipeline synchronization command to flush the command sequence through the graphics pipeline. The 3D pipeline will perform geometry processing for the 3D primitives. Once operations are complete, the resulting geometric objects are rasterized and the pixel engine colors the resulting pixels. Additional commands to control pixel shading and pixel back end operations may also be included for those operations.

In some embodiments, the graphics processor command sequence **1910** follows the media pipeline **1924** path when performing media operations. In general, the specific use and manner of programming for the media pipeline **1924** depends on the media or compute operations to be performed. Specific media decode operations may be offloaded to the media pipeline during media decode. In some embodiments, the media pipeline can also be bypassed and media decode can be performed in whole or in part using resources provided by one or more general purpose processing cores. In one embodiment, the media pipeline also includes elements for general-purpose graphics processor unit (GPGPU) operations, where the graphics processor is used to perform SIMD vector operations using computational shader programs that are not explicitly related to the rendering of graphics primitives.

In some embodiments, media pipeline **1924** is configured in a similar manner as the 3D pipeline **1922**. A set of commands to configure the media pipeline state **1940** are dispatched or placed into a command queue before the media object commands **1942**. In some embodiments, media

pipeline state commands **1940** include data to configure the media pipeline elements that will be used to process the media objects. This includes data to configure the video decode and video encode logic within the media pipeline, such as encode or decode format. In some embodiments, media pipeline state commands **1940** also support the use of one or more pointers to "indirect" state elements that contain a batch of state settings.

In some embodiments, media object commands **1942** supply pointers to media objects for processing by the media pipeline. The media objects include memory buffers containing video data to be processed. In some embodiments, all media pipeline states must be valid before issuing a media object command **1942**. Once the pipeline state is configured and media object commands **1942** are queued, the media pipeline **1924** is triggered via an execute command **1944** or an equivalent execute event (e.g., register write). Output from media pipeline **1924** may then be post processed by operations provided by the 3D pipeline **1922** or the media pipeline **1924**. In some embodiments, GPGPU operations are configured and executed in a similar manner as media operations.

Graphics Software Architecture

FIG. **20** illustrates exemplary graphics software architecture for a data processing system **2000** according to some embodiments. In some embodiments, software architecture includes a 3D graphics application **2010**, an operating system **2020**, and at least one processor **2030**. In some embodiments, processor **2030** includes a graphics processor **2032** and one or more general-purpose processor core(s) **2034**. The graphics application **2010** and operating system **2020** each execute in the system memory **2050** of the data processing system.

In some embodiments, 3D graphics application **2010** contains one or more shader programs including shader instructions **2012**. The shader language instructions may be in a high-level shader language, such as the High Level Shader Language (HLSL) or the OpenGL Shader Language (GLSL). The application also includes executable instructions **2014** in a machine language suitable for execution by the general-purpose processor core **2034**. The application also includes graphics objects **2016** defined by vertex data.

In some embodiments, operating system **2020** is a Microsoft® Windows® operating system from the Microsoft Corporation, a proprietary UNIX-like operating system, or an open source UNIX-like operating system using a variant of the Linux kernel. The operating system **2020** can support a graphics API **2022** such as the Direct3D API, the OpenGL API, or the Vulkan API. When the Direct3D API is in use, the operating system **2020** uses a front-end shader compiler **2024** to compile any shader instructions **2012** in HLSL into a lower-level shader language. The compilation may be a just-in-time (JIT) compilation or the application can perform shader pre-compilation. In some embodiments, high-level shaders are compiled into low-level shaders during the compilation of the 3D graphics application **2010**. In some embodiments, the shader instructions **2012** are provided in an intermediate form, such as a version of the Standard Portable Intermediate Representation (SPIR) used by the Vulkan API.

In some embodiments, user mode graphics driver **2026** contains a back-end shader compiler **2027** to convert the shader instructions **2012** into a hardware specific representation. When the OpenGL API is in use, shader instructions **2012** in the GLSL high-level language are passed to a user mode graphics driver **2026** for compilation. In some embodiments, user mode graphics driver **2026** uses operating system kernel mode functions **2028** to communicate with a kernel mode graphics driver **2029**. In some embodiments, kernel mode graphics driver **2029** communicates with graphics processor **2032** to dispatch commands and instructions.

IP Core Implementations

One or more aspects of at least one embodiment may be implemented by representative code stored on a machine-readable medium which represents and/or defines logic within an integrated circuit such as a processor. For example, the machine-readable medium may include instructions which represent various logic within the processor. When read by a machine, the instructions may cause the machine to fabricate the logic to perform the techniques described herein. Such representations, known as "IP cores," are reusable units of logic for an integrated circuit that may be stored on a tangible, machine-readable medium as a hardware model that describes the structure of the integrated circuit. The hardware model may be supplied to various customers or manufacturing facilities, which load the hardware model on fabrication machines that manufacture the integrated circuit. The integrated circuit may be fabricated such that the circuit performs operations described in association with any of the embodiments described herein.

FIG. **21** is a block diagram illustrating an IP core development system **2100** that may be used to manufacture an integrated circuit to perform operations according to an embodiment. The IP core development system **2100** may be used to generate modular, re-usable designs that can be incorporated into a larger design or used to construct an entire integrated circuit (e.g., an SOC integrated circuit). A design facility **2130** can generate a software simulation **2110** of an IP core design in a high level programming language (e.g., C/C++). The software simulation **2110** can be used to design, test, and verify the behavior of the IP core using a simulation model **2112**. The simulation model **2112** may include functional, behavioral, and/or timing simulations. A register transfer level (RTL) design **2115** can then be created or synthesized from the simulation model **2112**. The RTL design **2115** is an abstraction of the behavior of the integrated circuit that models the flow of digital signals between hardware registers, including the associated logic performed using the modeled digital signals. In addition to an RTL design **2115**, lower-level designs at the logic level or transistor level may also be created, designed, or synthesized. Thus, the particular details of the initial design and simulation may vary.

The RTL design **2115** or equivalent may be further synthesized by the design facility into a hardware model **2120**, which may be in a hardware description language (HDL), or some other representation of physical design data. The HDL may be further simulated or tested to verify the IP core design. The IP core design can be stored for delivery to a 3rd party fabrication facility **2165** using non-volatile memory **2140** (e.g., hard disk, flash memory, or any non-volatile storage medium). Alternatively, the IP core design may be transmitted (e.g., via the Internet) over a wired connection **2150** or wireless connection **2160**. The fabrication facility **2165** may then fabricate an integrated circuit that is based at least in part on the IP core design. The fabricated integrated circuit can be configured to perform operations in accordance with at least one embodiment described herein.

Exemplary System on a Chip Integrated Circuit

FIGS. **22-24** illustrated exemplary integrated circuits and associated graphics processors that may be fabricated using one or more IP cores, according to various embodiments

described herein. In addition to what is illustrated, other logic and circuits may be included, including additional graphics processors/cores, peripheral interface controllers, or general purpose processor cores.

FIG. **22** is a block diagram illustrating an exemplary system on a chip integrated circuit **2200** that may be fabricated using one or more IP cores, according to an embodiment. Exemplary integrated circuit **2200** includes one or more application processor(s) **2205** (e.g., CPUs), at least one graphics processor **2210**, and may additionally include an image processor **2215** and/or a video processor **2220**, any of which may be a modular IP core from the same or multiple different design facilities. Integrated circuit **2200** includes peripheral or bus logic including a USB controller **2225**, UART controller **2230**, an SPI/SDIO controller **2235**, and an I^2S/I^2C controller **2240**. Additionally, the integrated circuit can include a display device **2245** coupled to one or more of a high-definition multimedia interface (HDMI) controller **2250** and a mobile industry processor interface (MIPI) display interface **2255**. Storage may be provided by a flash memory subsystem **2260** including flash memory and a flash memory controller. Memory interface may be provided via a memory controller **2265** for access to SDRAM or SRAM memory devices. Some integrated circuits additionally include an embedded security engine **2270**.

FIG. **23** is a block diagram illustrating an exemplary graphics processor **2310** of a system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment. Graphics processor **2310** can be a variant of the graphics processor **2210** of FIG. **22**. Graphics processor **2310** includes a vertex processor **2305** and one or more fragment processor(s) **2315**A-**2315**N (e.g., **2315**A, **2315**B, **2315**C, **2315**D, through **2315**N-**1**, and **2315**N). Graphics processor **2310** can execute different shader programs via separate logic, such that the vertex processor **2305** is optimized to execute operations for vertex shader programs, while the one or more fragment processor(s) **2315**A-**2315**N execute fragment (e.g., pixel) shading operations for fragment or pixel shader programs. The vertex processor **2305** performs the vertex processing stage of the 3D graphics pipeline and generates primitives and vertex data. The fragment processor(s) **2315**A-**2315**N use the primitive and vertex data generated by the vertex processor **2305** to produce a framebuffer that is displayed on a display device. In one embodiment, the fragment processor(s) **2315**A-**2315**N are optimized to execute fragment shader programs as provided for in the OpenGL API, which may be used to perform similar operations as a pixel shader program as provided for in the Direct 3D API.

Graphics processor **2310** additionally includes one or more memory management units (MMUs) **2320**A-**2320**B, cache(s) **2325**A-**2325**B, and circuit interconnect(s) **2330**A-**2330**B. The one or more MMU(s) **2320**A-**2320**B provide for virtual to physical address mapping for integrated circuit **2310**, including for the vertex processor **2305** and/or fragment processor(s) **2315**A-**2315**N, which may reference vertex or image/texture data stored in memory, in addition to vertex or image/texture data stored in the one or more cache(s) **2325**A-**2325**B. In one embodiment the one or more MMU(s) **2325**A-**2325**B may be synchronized with other MMUs within the system, including one or more MMUs associated with the one or more application processor(s) **2205**, image processor **2215**, and/or video processor **2220** of FIG. **22**, such that each processor **2205**-**2220** can participate in a shared or unified virtual memory system. The one or more circuit interconnect(s) **2330**A-**2330**B enable graphics processor **2310** to interface with other IP cores within the SoC, either via an internal bus of the SoC or via a direct connection, according to embodiments.

FIG. **24** is a block diagram illustrating an additional exemplary graphics processor **2410** of a system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment. Graphics processor **2410** can be a variant of the graphics processor **2210** of FIG. **22**. Graphics processor **2410** includes the one or more MMU(s) **2320**A-**2320**B, caches **2325**A-**2325**B, and circuit interconnects **2330**A-**2330**B of the integrated circuit **2300** of FIG. **23**.

Graphics processor **2410** includes one or more shader core(s) **2415**A-**2415**N (e.g., **2415**A, **2415**B, **2415**C, **2415**D, **2415**E, **2415**F, through **2315**N-**1**, and **2315**N), which provides for a unified shader core architecture in which a single core or type or core can execute all types of programmable shader code, including shader program code to implement vertex shaders, fragment shaders, and/or compute shaders. The exact number of shader cores present can vary among embodiments and implementations. Additionally, graphics processor **2410** includes an inter-core task manager **2405**, which acts as a thread dispatcher to dispatch execution threads to one or more shader cores **2415**A-**2415**N and a tiling unit **2418** to accelerate tiling operations for tile-based rendering, in which rendering operations for a scene are subdivided in image space, for example to exploit local spatial coherence within a scene or to optimize use of internal caches.

The following examples pertain to further embodiments.

In one example, an apparatus comprises a memory array having a plurality of storage segments. The plurality of storage segments may include: a plurality of bitcells each to store a data bit; a first local bitline coupled to a first collection of the plurality of bitcells to output read data from the first collection of the plurality of bitcells; a second local bitline coupled to a second collection of the plurality of bitcells to output read data from the second collection of the plurality of bitcells; and a latch circuit having a first input coupled to the first local bitline and a second input coupled to the second local bitline, the latch circuit to output a segment output, the segment output comprising a static signal.

In an example, the memory array comprises a register file having a plurality of entries, where each of the plurality of storage segments includes a portion of the plurality of entries.

In an example, the memory array further comprises a static logic circuit to receive the segment output from the plurality of storage segments and to output read data based on the segment output from the plurality of storage segments.

In an example, the apparatus further comprises a mapping circuit to store first information associated with a first thread having a first priority in a first storage segment of the plurality of storage segments and store second information associated with a second thread having a second priority in a second storage segment of the plurality of storage segments, the first priority higher than the second priority, and where the first storage segment is closer to the static logic circuit than the second storage segment.

In an example, the apparatus further comprises a plurality of transmission interconnects to couple the plurality of storage segments to the static logic circuit, where at least some of the plurality of transmission interconnects have differing lengths.

In an example, the plurality of transmission interconnects comprises a segmented global bitline.

In an example, the segmented global bitline is to provide the read data to the static logic circuit without a pre-charge operation on the segmented global bitline.

In an example, the apparatus further comprises a mapping circuit to cause more frequently accessed data to be stored in one or more of the plurality of storage segments coupled to the static logic circuit by a transmission interconnect having a shorter length than at least one other of the plurality of transmission interconnects.

In an example, the static logic circuit comprises: a first selection circuit coupled to a first portion of the plurality of storage segments to output a first read value; a second selection circuit coupled to a second portion of the plurality of storage segments to output a second read value; and a third selection circuit coupled to an output of the first selection circuit and to an output of the select second selection circuit to output the read data from one of the first read value and the second read value.

In an example, the memory array is to consume a same amount of power to read a logic one value as to read a logic zero value.

In another example, a method comprises: identifying a read frequency of data to be stored in a memory; selecting a segment of a plurality of segments of the memory for storage of the data based at least in part on the read frequency; and storing the data in the selected segment of the plurality of segments.

In an example, the method further comprises: selecting a first segment of the plurality of segments for storing first data associated with a first thread having a first priority; and selecting a second segment of the plurality of storage segments for storing second data associated with a second thread having a second priority, the first priority higher than the second priority, and where the first segment has a reduced distance to a static logic circuit of the memory than the second segment.

In an example, the method further comprises receiving the data to be stored in the memory within a controller of the memory and identifying the read frequency in the controller of the memory based at least in part on a priority indicator associated with the data.

In an example, the method further comprises re-mapping a first location of the memory identified in a write request to a second location of the memory associated with the selected segment.

In an example, the method further comprises reading the data from the selected segment without pre-charging a segmented global bitline on which the data is output.

In an example, the method further comprises identifying the read frequency in a compiler.

In yet another example, a processor comprises at least one processor core and at least one graphics processor. The at least one graphics processor comprises a register file comprising a plurality of storage segments to store information and output a plurality of segment outputs via a plurality of segmented bitlines to a static logic circuit to receive the plurality of segment outputs from the plurality of storage segments and to output read data based on the plurality of segment outputs, where the register file is to output the read data with a same amount of power without regard to a logic state of the read data.

In an example, the plurality of storage segments comprises: a plurality of bitcells each to store a data bit; a first local bitline coupled to a first collection of the plurality of bitcells to output first read data from the first collection of the plurality of bitcells; a second local bitline coupled to a second collection of the plurality of bitcells to output second read data from the second collection of the plurality of bitcells; and a latch circuit having a first input coupled to the first local bitline and a second input coupled to the second local bitline, the latch circuit to output a segment output via one of the plurality of segmented bitlines, the segment output comprising a static signal.

In an example, the at least one graphics processor further comprises a mapping circuit to store first information associated with a first thread having a first priority in a first storage segment of the plurality of storage segments and store second information associated with a second thread having a second priority in a second storage segment of the plurality of storage segments, the first priority higher than the second priority, and where the first storage segment is closer to the static logic circuit than the second storage segment.

In an example, the register file further comprises a plurality of transmission interconnects to couple the plurality of storage segments to the static logic circuit, where at least some of the plurality of transmission interconnects have differing lengths, where the plurality of transmission interconnects comprises a segmented global bitline, and where the segmented global bitline is to provide the plurality of segment outputs to the static logic circuit without a pre-charge operation on the segmented global bitline.

In a still further example, an apparatus comprises: means for identifying a read frequency of data to be stored in a memory means; means for selecting a segment of a plurality of segments of the memory means for storage of the data based at least in part on the read frequency; and means for storing the data in the selected segment of the plurality of segments.

In an example, the apparatus further comprises: means for selecting a first segment of the plurality of segments for storing first data associated with a first thread having a first priority; and means for selecting a second segment of the plurality of storage segments for storing second data associated with a second thread having a second priority, the first priority higher than the second priority, and where the first segment has a reduced distance to a static logic circuit of the memory means than the second segment.

In an example, the apparatus further comprises means for receiving the data to be stored in the memory means within a controller of the memory means and identifying the read frequency in the controller of the memory means based at least in part on a priority indicator associated with the data.

In an example, the apparatus further comprises means for re-mapping a first location of the memory means identified in a write request to a second location of the memory means associated with the selected segment.

Understand that various combinations of the above examples are possible.

Note that the terms "circuit" and "circuitry" are used interchangeably herein. As used herein, these terms and the term "logic" are used to refer to alone or in any combination, analog circuitry, digital circuitry, hard wired circuitry, programmable circuitry, processor circuitry, microcontroller circuitry, hardware logic circuitry, state machine circuitry and/or any other type of physical hardware component. Embodiments may be used in many different types of systems. For example, in one embodiment a communication device can be arranged to perform the various methods and techniques described herein. Of course, the scope of the present invention is not limited to a communication device, and instead other embodiments can be directed to other types of apparatus for processing instructions, or one or more machine readable media including instructions that in

response to being executed on a computing device, cause the device to carry out one or more of the methods and techniques described herein.

Embodiments may be implemented in code and may be stored on a non-transitory storage medium having stored thereon instructions which can be used to program a system to perform the instructions. Embodiments also may be implemented in data and may be stored on a non-transitory storage medium, which if used by at least one machine, causes the at least one machine to fabricate at least one integrated circuit to perform one or more operations. Still further embodiments may be implemented in a computer readable storage medium including information that, when manufactured into a SoC or other processor, is to configure the SoC or other processor to perform one or more operations. The storage medium may include, but is not limited to, any type of disk including floppy disks, optical disks, solid state drives (SSDs), compact disk read-only memories (CD-ROMs), compact disk rewritables (CD-RWs), and magneto-optical disks, semiconductor devices such as read-only memories (ROMs), random access memories (RAMs) such as dynamic random access memories (DRAMs), static random access memories (SRAMs), erasable programmable read-only memories (EPROMs), flash memories, electrically erasable programmable read-only memories (EEPROMs), magnetic or optical cards, or any other type of media suitable for storing electronic instructions.

While the present invention has been described with respect to a limited number of embodiments, those skilled in the art will appreciate numerous modifications and variations therefrom. It is intended that the appended claims cover all such modifications and variations as fall within the true spirit and scope of this present invention.

What is claimed is:

1. An apparatus comprising:

a memory array having a plurality of storage segments, the plurality of storage segments including:

a plurality of bitcells each to store a data bit;

a first local bitline coupled to a first collection of the plurality of bitcells to output read data from the first collection of the plurality of bitcells;

a second local bitline coupled to a second collection of the plurality of bitcells to output read data from the second collection of the plurality of bitcells; and

a latch circuit having a first input coupled to the first local bitline and a second input coupled to the second local bitline, the latch circuit to output a segment output, the segment output comprising a static signal.

2. The apparatus of claim **1**, wherein the memory array comprises a register file having a plurality of entries, wherein each of the plurality of storage segments includes a portion of the plurality of entries.

3. The apparatus of claim **1**, wherein the memory array further comprises a static logic circuit to receive the segment output from the plurality of storage segments and to output read data based on the segment output from the plurality of storage segments.

4. The apparatus of claim **3**, further comprising a mapping circuit to store first information associated with a first thread having a first priority in a first storage segment of the plurality of storage segments and store second information associated with a second thread having a second priority in a second storage segment of the plurality of storage segments, the first priority higher than the second priority, and wherein the first storage segment is closer to the static logic circuit than the second storage segment.

5. The apparatus of claim **3**, further comprising a plurality of transmission interconnects to couple the plurality of storage segments to the static logic circuit, wherein at least some of the plurality of transmission interconnects have differing lengths.

6. The apparatus of claim **5**, wherein the plurality of transmission interconnects comprises a segmented global bitline.

7. The apparatus of claim **6**, wherein the segmented global bitline is to provide the read data to the static logic circuit without a pre-charge operation on the segmented global bitline.

8. The apparatus of claim **5**, further comprising a mapping circuit to cause more frequently accessed data to be stored in one or more of the plurality of storage segments coupled to the static logic circuit by a transmission interconnect having a shorter length than at least one other of the plurality of transmission interconnects.

9. The apparatus of claim **3**, wherein the static logic circuit comprises:

a first selection circuit coupled to a first portion of the plurality of storage segments to output a first read value;

a second selection circuit coupled to a second portion of the plurality of storage segments to output a second read value; and

a third selection circuit coupled to an output of the first selection circuit and to an output of the second selection circuit to output the read data from one of the first read value and the second read value.

10. The apparatus of claim **1**, wherein the memory array is to consume a same amount of power to read a logic one value as to read a logic zero value.

11. A machine-readable medium having stored thereon instructions, which if performed by a machine cause the machine to perform a method comprising:

identifying a read frequency of data to be stored in a memory;

selecting a segment of a plurality of segments of the memory for storage of the data based at least in part on the read frequency; and

storing the data in the selected segment of the plurality of segments.

12. The machine-readable medium of claim **11**, wherein the method further comprises:

selecting a first segment of the plurality of segments for storing first data associated with a first thread having a first priority; and

selecting a second segment of the plurality of storage segments for storing second data associated with a second thread having a second priority, the first priority higher than the second priority, and wherein the first segment has a reduced distance to a static logic circuit of the memory than the second segment.

13. The machine-readable medium of claim **11**, wherein the method further comprises receiving the data to be stored in the memory within a controller of the memory and identifying the read frequency in the controller of the memory based at least in part on a priority indicator associated with the data.

14. The machine-readable medium of claim **11**, wherein the method further comprises re-mapping a first location of the memory identified in a write request to a second location of the memory associated with the selected segment.

15. The machine-readable medium of claim **11**, wherein the method further comprises reading the data from the

selected segment without pre-charging a segmented global bitline on which the data is output.

16. The machine-readable medium of claim **11**, wherein the method further comprises identifying the read frequency in a compiler.

17. A processor comprising:

at least one processor core; and

at least one graphics processor, wherein the at least one graphics processor comprises a register file comprising a plurality of storage segments to store information and output a plurality of segment outputs via a plurality of segmented bitlines to a static logic circuit to receive the plurality of segment outputs from the plurality of storage segments and to output read data based on the plurality of segment outputs, wherein the register file is to output the read data with a same amount of power without regard to a logic state of the read data.

18. The processor of claim **17**, wherein the plurality of storage segments comprises:

a plurality of bitcells each to store a data bit;

a first local bitline coupled to a first collection of the plurality of bitcells to output first read data from the first collection of the plurality of bitcells;

a second local bitline coupled to a second collection of the plurality of bitcells to output second read data from the second collection of the plurality of bitcells; and

a latch circuit having a first input coupled to the first local bitline and a second input coupled to the second local bitline, the latch circuit to output a segment output via one of the plurality of segmented bitlines, the segment output comprising a static signal.

19. The processor of claim **17**, wherein the at least one graphics processor further comprises a mapping circuit to store first information associated with a first thread having a first priority in a first storage segment of the plurality of storage segments and store second information associated with a second thread having a second priority in a second storage segment of the plurality of storage segments, the first priority higher than the second priority, and wherein the first storage segment is closer to the static logic circuit than the second storage segment.

20. The processor of claim **17**, wherein the register file further comprises a plurality of transmission interconnects to couple the plurality of storage segments to the static logic circuit, wherein at least some of the plurality of transmission interconnects have differing lengths, wherein the plurality of transmission interconnects comprises a segmented global bitline, and wherein the segmented global bitline is to provide the plurality of segment outputs to the static logic circuit without a pre-charge operation on the segmented global bitline.

* * * * *